UNIVERSITY OF OSLO
Department of Informatics

Visualizing Multi-Way Sensory Data

Master thesis

Henning Risvik

May 2, 2008

Abstract

This thesis is part of a project called PanelCheck which involves creation of a software tool. A statistical method called Manhattan plot has been implemented in this software tool, which is applied for visualization of performance of assessors from sensory panels. There is background on the methodology of the Manhattan plot and an example on usage is presented using sensory data of a cheese experiment. There is information about the software packages used for creation of the application and information on the design of the application. This together with examples on usage of the software packages for getting started on development for PanelCheck, more specifically for getting started on implementation of a new plot module. Additionally there has been focus on efficiency, especially concerning numerical computing.

Acknowledgements

First I would like to thank my supervisors Hans Petter Langtangen, Tormod Næs and Oliver Tomic for all the guidance. A special thanks goes to Oliver Tomic for all the constructive critic, the many discussions, the many suggestions and taking the time for all this even though far away for the main work period (I must say Skype has served very well for communication between Australia and Norway). And also for suggesting me to continue as developer for PanelCheck after the student project was over, back in 2005. Thanks to all of the PanelCheck team. Thanks to the University of Oslo for interesting courses and to students for collaboration. Thanks to my friends Johan Laake and Martin Omnes for proofreading. Finally, thanks to my family for their support.

Contents

List of Figures

Chapter 1

Introduction

This introduction will give information about the assignment at hand, information about the employer and the project which this thesis is a part of. The thesis has been a collaborative project between the University of Oslo and the research institute Matforsk. The project at Matforsk involves creation and development of a software tool. In this introduction there is information about this tool and the motivation for its development.

1.1 Matforsk

Matforsk - Norwegian Food Research Institute - is a leading research center in the understanding of food quality. Matforsk's primary goal is to contribute to increased profitability in the food industry through food research and development at an internationally recognized level.

The organization was established in 1970 and today it has about 160 employees. It is financed by a research levy from the industry, public funding and contract work, and is owned by the Agricultural Food Research Foundation, which is an independent, non-profit making organization.

Matforsk focuses its research and development activities in order to be internationally competitive, and has binding commitments with other research institutions.

1.2 Sensory Panel

Sensory analysis is one of many areas of research at Matforsk. An important way of gathering data in this research area is done by sensory descriptive analysis. This is carried out by a sensory panel.

At Matforsk there are 12 employees with positions as assessors of Matforsk's sensory panel. They do sensory descriptive analysis regularly and have on average 12 years of experience. Training and sensory descriptive

Figure 1.1: Matforsk AS, research institute at Ås

analysis three times per week makes this sensory panel among the best in Europe.

This type of panel usually consists of 4-12 individuals, which carry out the sensory testing. Such testing yield sensory profiling data, which there will be more on in chapter 2 on page 9.

With sensory descriptive analysis we mean something that has to do with sense perception, i.e. how humans describe a product's appearance, odor, flavor or texture. Sensory descriptive analysis can be done on all kinds of products, from food, perfume to for example speakers or other home cinema equipment. This is possible on any product that can be described by sense perception in some way.

There are various ways of carrying out such an analysis. Here is a common method used at Matforsk:

Descriptive test is used for describing which characteristics that makes up the total perception of a product and the score values of these attributes. This can be interesting in relation to product development, studies of changes in production, definition of standards and e.g. in comparing with competitive products. These tests are often used when one know there are clear differences between the products.

Sensory descriptive analysis is a very efficient and an important way to control products, because humans (a sensory panel) do the actual sensory testing of the products, and these products are, after all, made for humans.

Measuring the performance of a panel is possible with usage of statistical methods, and on this matter is where PanelCheck comes into the picture.

1.3 The PanelCheck Project

PanelCheck is an ongoing project at Matforsk which is funded by the Norwegian Research Council. Within the project Matforsk has close collaboration with Danish research institutes of the same area (University of Copenhagen and DTU).

The PanelCheck project is about research and development on statistical methods that are considered for implementation in a software tool. The creation of the software tool started with the beginning of the project in 2005. The name of the software tool is "PanelCheck".

The project has four main problem areas:

1. Development of simple methods, and a minimum of these tools, for panel monitoring.
2. Simplification of advanced methods for panel monitoring. This involves investigation of advantages/disadvantages and research on how to make them sufficiently user-friendly. The methods can be model-based and one will be looking at all of the available information at once (multivariate analysis). For that reason usage of these methods may have important advantages over "simpler" methods.
3. Investigation on how to utilize information of methods in a best possible way.
4. Development of an "easy-to-use" software tool for visualizations and viewing of results of the statistical methods.

1.3.1 Motivation

Every sensory panel needs training. There exist a large number of statistical methods for performance control of the assessors, but most of the methods are complex and hard to use for non-statisticians. This is one of the main reasons for the creation of the PanelCheck software. It should aid well in easy and fast analysis of the sensory data and thus the performance of the sensory panel.

Matforsk offers the software for free to anyone who wishes to use it. A Windows installer is available on the project web site. It is free because Matforsk wants to reach out to as many potential users as possible in Norway and internationally. This is giving Matforsk an opportunity to market their competence world wide. On short term Matforsk wishes to earn revenues by holding courses on how to use the software tool efficiently and on background knowledge of the statistics involved. On the long term Matforsk

wishes to establish themselves as an even stronger internationally leading community in the areas of statistics and sensory analysis.

Finally, as the ultimate and ambitious goal for the project, which PanelCheck is a small piece of for Matforsk:

To contribute to

- increased profitability in the food industry
- improved food quality
- safer, healthier and longer lasting food

1.3.2 General Information

The projects in Norway and Denmark are two parallel projects cooperating on development in the same scientific field. Both in Norway and Denmark development and research is done on statistical methods of interest for implementation in the software tool. The time span for the project is set to be for 4 years. It was started at beginning of 2005 and will last until 2009.

In Norway at Matforsk there are 4 researchers and 1 post. doc. working on the project, as well as myself as external staff for additional software programming. Matforsk receives for each of the four years 1.1 million NOK from the Norwegian Research Council. Matforsk itself contributes with about 350.000 NOK, in form of working hours, to the project each year. Partners of the industry contribute with about 420.000 NOK each year, also in the form of working hours. Working hours include testing of software, new suggestions and ideas for future versions and testing of the statistical methods implemented. The industrial partners of the project in Norway are:

- **Arcus AS** - Supplier of wine and spirits
- **Fiskeriforskning** - Research institute that work for the fishery and aquaculture industry
- **Gilde** - Supplier of meat products
- **Hennig Olsen-Is** - Supplier of ice cream products
- **Tine** - Norway's leading supplier of food products

In Denmark the "parallel" project consists of 1 researcher of the University of Copenhagen and, at DTU, 1 researcher and 1 PhD student. The industrial partners of the project in Denmark are (note: not only of the food industry):

- **Bang & Olufsen** - Supplier of exclusive high-end audio and video products
- **Chew Tech** - Specialist in all aspects of chewing gum research and development
- **Chr. Hansen** - Supplier of natural ingredient solutions
- **Danish Institute for Fisheries Research** - Fisheries research with the purpose of giving advice on sustainable utilization of the living marine and freshwater resources
- **Danish Meat Research Institute** - Leading knowledge center within meat and slaughter technology

PanelCheck is published under the GNU General Public License (see reference [35] to read more about the license). It is an Open-Source software which means that the source code can be read and modified by anyone. For now PanelCheck is mainly developed for Windows operating systems (untested on Unix-like and Mac systems), but the source code and most of the packages used in the software are platform independent. There is one platform dependent package, but it can be excluded. The only functionality that will be lost is reading of Excel files and creation of PowerPoint files.

The PanelCheck software is used by many companies and institutes world wide. Here are some numbers from statistics of the project web-site at the end of December 2007:

- Downloads (Windows installer): 1147
- Downloads (Python source code): 272
- Web Traffic (unique visits): 2196

The PanelCheck users:

- Registered Users: 388
- Countries: 49
- Organizations: 192
- Intitutes/gov.: 43
- Universities: 50
- Commercial: 99

So, it is expected that all calculations are error-free and that the software is stable and fast. For this to be possible, together with low development costs and fast development, it is important with a good underlying "engine".

PanelCheck is built on the programming language Python, a robust and reliable "engine" for a very wide range of software systems. More can be read about Python and the packages used in chapter 4 on page 24.

1.3.3 Plot Methods

PanelCheck is a software tool for analysis of the sensory data. It can evaluate the panel of assessors as one, and you can think of it as a tool for calibration of the sensory panel. Visualizations, of many different statistical methods, are created fast and analysis can begin. These plots are also meant to be readable to people without deep knowledge of statistics.

Today PanelCheck has a number of standard methods for analysis. PanelCheck is meant to be limited to some good, different and well tested methods. These are the current plotting methods that exists in PanelCheck (version 1.3.0):

Univariate methods:

- **Line Plots** - Raw data score values.
- **Mean & STD plot** - Mean and standard deviation of raw data.
- **Correlation Plots** - See article [4].
- **Profile Plots** - See article [4].
- **Eggshell Plots** - See article [4].
- **F & p Plots** - Data of one-way ANOVA. See article [4].
- **MSE Plots** - Data of one-way ANOVA. See article [4].
- **p-MSE Plots** - Data of one-way ANOVA. See article [4].

Multivariate methods:

- **Tucker-1 Plots** - See article [9].
- **Manhattan Plots** - More information later on (or see article [3]).
- **Consensus Plots** - Principal Component Analysis of data of consensus of assessors with averaged replicates [6]. The STATIS method is also included [10].

- **Mixed Model ANOVA Plots** - See article [11].

Each plot reveals some important information about the data set. Together, a number of plots, can give a good overview of one assessor, the whole sensory panel as one or whether the assessors of the sensory panel functions well together or not.

The program is also to be extended with additional methods which will give information that the currently implemented methods cannot show. But a new method is always evaluated and considered heavily before implementing it. In addition there can also be different ways of carrying out and visualizing one statistical method. That is a problem area which also must be discussed before implementing.

1.4 Thesis

This thesis is about improving and adding possibilities to the Open-Source software tool PanelCheck. The main part is the implementation of the Manhattan plot in PanelCheck. I have also gone into some investigation on the functionality and usability of this statistical method.

Here are a few main points of the thesis that were written down prior to project start:

- Analysis of 3-way data from descriptive sensory tests using new multivariate statistical methods.
- Visualization of results from analysis.
- Analysis and testing of method.

Manhattan plot is a statistical method for analysis of multivariate data. In PanelCheck Manhattan plot will be used as a screening tool for leaders/-managers of sensory panels. The method has a certain type of visualization which has been implemented. The statistical method, with the visualization, has been implemented as a standard plot module in PanelCheck.

The Manhattan plot requires usage of a technique called the Principal Component Analysis (PCA). I decided to create a PCA module for Python from scratch. Methodology of the PCA and of the Manhattan plot is covered to some extent, see chapter 3 on page 13.

Furthermore, there are details on "How to implement a plot module in PanelCheck". Thus technical details of the PanelCheck software have also been covered. On the software side PanelCheck is not well documented so I have made diagrams and written an overview to help in understanding of the design. I have also gone through the different software packages used

in PanelCheck with a little about usage of the main packages for getting started on development.

Another interesting part of this thesis is the usage of Python for numerical computing. During the creation of the PCA module for Python I investigated performance of a few different techniques concerning numerical computing in Python (here with serial computing only).

Chapter 2

Sensory Profiling Data

Sensory profiling data is the result of score values given by a sensory panel through sensory description. The score values are given on certain attributes that are meant to describe a product. The job of the sensory panel is to carry out sensory descriptive analysis for different product(s). The choosing of attributes that describe the product(s) is very important, and might matter more than the performance of the panel. See ISO standard [37], in the references, on selection of attributes for descriptive tests.

2.1 Structure

The sensory data sets gathered at Matforsk are multi-way data sets. For each data set there are K assessors, M attributes and N samples, thus 3-way data. The data consist of score values given on a scale from 1-9, at Matforsk. Other organizations use different scales (two examples are food science Australia: 0-100 and the University of Copenhagen: 1-15). Each attribute could also have its own scale (in that case standardization might be very important for some types of analysis).

A usual number of attributes is around 5-10 and good attribute choices will describe the product well. At Matforsk they use 15-30 attributes for most of their data sets. That many attributes can describe the product better, but even greater knowledge of the product is required for making good and significant attribute choices.

Usually the data sets have more than one replicate (2, 3 or more). That is also something desirable, because with more than one replicate one can better see if the assessor is able to repeat him- or herself.

Figure 2.1 shows an example of how sensory profiling data may be structured. One table in this figure contains score values given by one assessor for each attribute on each sample. In total, 4 assessors, 2 replicates, 8 samples and 5 attributes. Of this figure you can also get the sense of what is meant by multi-way sensory data. This data set is said to be 3-way data,

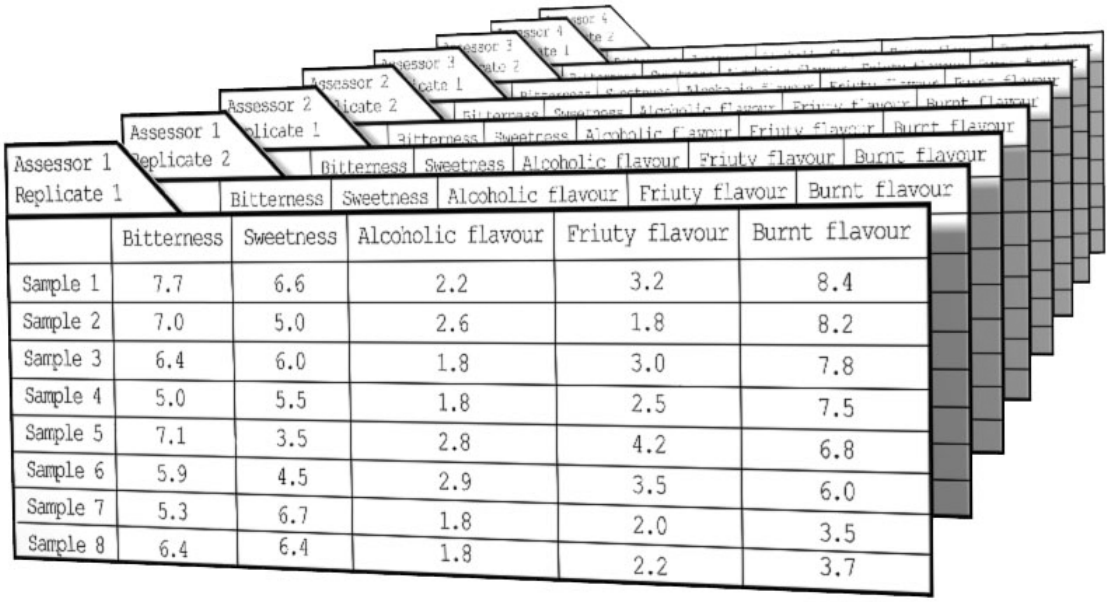

Figure 2.1: A beer test, scores are arbitrary values

but technically it is a 4-way data set if we take into account that we have 2 x replicates also. But this is not always the case and in analysis the average of the replicates is often used. So the term 3-way data is appropriate.

2.2 Missing Values & Unbalanced Data

Sensory profiling tests can last for more than one day. With 10 assessors, or more, in the sensory panel it is not uncommon that one or more of the assessors do not complete all the tests. This would yield an unbalanced data set (i.e. some samples, with replicates, can be missing for certain assessors). Another case of incomplete data is missing values. Certain score values might be missing for various reasons. Random missing data means there are missing values in the data set and this can be the result of, something that is most likely, human mistakes (e.g. forgotten entry or erroneous entry) or an error in the system of handling the data set. Some companies have automated systems (such as Compusense, Fizz or Logic8) for handling the process of gathering data and structuring the data afterwards. Such systems increase the chance of ending up with a complete data set.

One would want to keep as much data as possible, because sensory descriptive analysis is usually a quite cost consuming process. With many assessors (specially trained) doing sensory descriptive analysis on many samples, possibly over various days, we can conclude that data is valuable and that we do not want to be throwing any of it away. So, the software tool for analysis of the sensory data should be able to handle unbalanced data and missing values somehow. Both cases will be noticed by PanelCheck and handled in a proper manner. Either by a general method for the situation or by user input, so the user choose the best solution.

Handling in PanelCheck

An unbalanced data set can be all from multiple missing samples for multiple assessors to a single missing replicate (meaning one row of scores) for one assessor. In any case, the unbalanced data set will have to be balanced out for use in most of the methods for analysis in PanelCheck. One way of balancing the data is to remove the samples affected for all assessors. Another way (the other way around) is to remove the assessors that have not completed all the tests.

For unbalanced data PanelCheck will give the user options on what to remove from the data set for analysis.

If there are not too many missing values fairly good new values can be imputed instead of the missing ones. There are different techniques for calculation of the new values. A simple example would be to compute the mean over the scores of all remaining samples for that particular attribute.

PanelCheck will give information about missing data, if present, and calculate new values, by zero-th order imputation, automatically.

2.3 The Cheese Data Set

The sensory data of the cheese data set was taken from measurements of a cheese type called Norvegia (a popular cheese type in Norway, made by Tine). The experiment was carried out at Matforsk. The main question to be answered of the analysis was: What color packaging of the cheese would store it best? The cheese type has been stored in different conditions for 36 hours before gathering of the sensory data. The different conditions was:

- exposure to light through colored plastic films of the colors: red, yellow, green, blue and violet
- exposure to light through a transparent plastic film (white light)
- without exposure to light (dark room)

Sample number	Oxygen level (%)	Type of light exposure
1	0	Blue
2	0	Violet
3	0	Green
4	0	Yellow
5	0	Red
6	0	Transparent
7	0	Dark/ no light exposure
8	1	Blue
9	1	Violet
10	1	Green
11	1	Yellow
12	1	Red
13	1	Transparent
14	1	Dark/no light exposure

Table 2.1: The samples of the cheese data set.

There are 7 different conditions mentioned above. For each of these 7 conditions there was also another factor: the level of remaining oxygen in the package of cheese. There was one group with an oxygen level at 0% and another group with an oxygen level at 1%. Thus, in total, the data set consist of 14 samples. See table 2.1 for the complete list (this is table 2 from article [3]).

The attributes were chosen by experts at Matforsk. They have knowledge of the product and its reaction to light. Score values could, in this case, be given on the attributes by sensory description. In total there are 17 attributes and they are such as various flavors and odors.

The score values (sensory data) were given by a sensory panel of 12 assessors. Of these assessors there were 9 with much experience (trained), assessors 1-9, and 3 with little experience (untrained), assessors 10-12.

The data set has 2 replicates for each sample and they were presented in a randomized order to the assessors.

Chapter 3

Methodology

One fundamental part of the Manhattan plot is the Principal Component Analysis (PCA). PCA is an important tool in multivariate analysis. In this chapter we will go through some of the theory behind and about this technique. Mathematically PCA is actually an estimation of eigenvectors. This chapter presents one algorithm for finding eigenvectors, the NIPALS algorithm. First, some information about preprocessing of data:

3.1 Preprocessing

The preprocessing part can make the difference between a useful model and no model at all.

One term for preprocessing is called autoscaling, which is the combination of mean centering and standardization. Standardization is one type of scaling where each value is scaled by 1/STD. In our cheese data set we know that all the scores lies in the range 1-9, so for this data set standardization should not be necessary. For most of the PCA-based methods of PanelCheck autoscaling is possible.

We center each element $X_{(i,j)}$ by this formula:

$$X_{centered(i,j)} = X_{(i,j)} - \overline{X}_{(i)} \tag{3.1}$$

In equation 3.1 i is index of a column and j is index of a row and $\overline{X}_{(i)}$ means the average of attribute vector i of X.

We standardize X by this formula:

$$X_{standardized(i,j)} = (X_{(i,j)} - \overline{X}_{(i)})/STD(X_{(i)}) \tag{3.2}$$

In equation 3.2 i is index of a column and j is index of a row and $STD(X_{(i)})$ means the standard deviation of attribute vector i and $\overline{X}_{(i)}$ means

the average of this attribute vector.

One of these formulas are used on a data matrix X prior to PCA. In the Manhattan plot preprocessing is done for each assessor, then PCA is carried out for each of the preprocessed data matrices.

3.2 Principal Component Analysis

A short description of the technique is: simplifying a data set by reducing multidimensional data sets to lower dimensions for analysis.

In this description of the Principal Component Analysis (PCA) we will be looking at an example data set for simplifying explanations, the cheese data set mentioned in the preceding chapter.

3.2.1 PC Model

The method involves decomposing a data matrix X into a structure part and a noise part. Later on there will be more on the scores (T), the loadings (P) and the residuals (E).

The PC model is the matrix product TP^T (the structure):

$$X = TP^T + E = Structure + Noise \tag{3.3}$$

In equation 3.3 we can see how T (scores) and P (loadings) form the TP (structure) part of the equation. Figure 3.1 shows how principal components are represented in equation 3.3. This summation will get us back to X, but what we are interested in is each of the scores and loadings which are collected in TP^T, and for the Manhattan plot the residuals. We assume that X can be split into the sum of the matrix product TP^T and the residual matrix E and the idea is to use T and P instead of X.

The first principal component (PC) will stretch out in the direction with the largest variance, in variable space, and form the first PC axis. The second PC will stretch out in the direction orthogonal to the first PC axis and with the second largest variance, and form the second PC axis. The third PC stretches out in the direction with third largest variance, and so on for the remaining PCs. All PC axes are orthogonal to each other and they will each cross each other where the average object of our data of variable space is. That average object will be the origo of the new PC space, the model center. Figure 3.2 illustrates how a PC space can be formed.

About the number of PCs, there is a maximum, and that is the least of N or M (min(N,M)) of the data matrix. We have 14 samples (objects) and

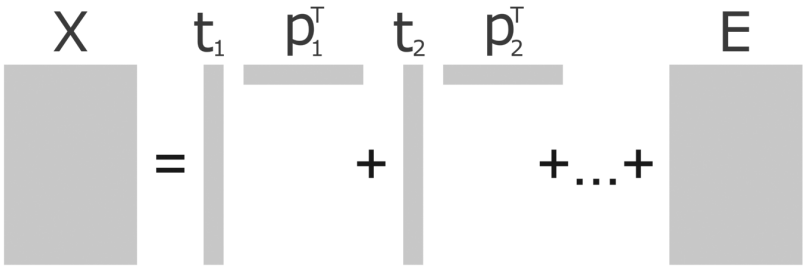

Figure 3.1: Illustration showing collection of scores and loadings for each PC.

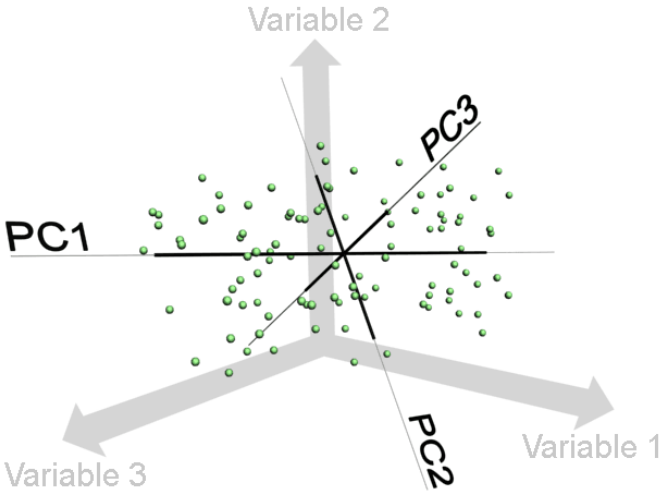

Figure 3.2: Illustration of change to PC coordinate system.

17 attributes (variables) if we focus on one assessor of our cheese data set. N=14 and M=17 means there can be a maximum of 14 PCs. If we use all 14 we will have a full model, but then we have only replaced the variable space with a new coordinate system, the PC-space, and there has been no separation, so E will be equal to 0. The separation of the structure part and the noise part is one of the strengths of a PCA.

We are typically most interested in the first few PCs (PC_1 and PC_2), where there is most explained variance. We want to separate the structure and the noise and at the same time we want there to be high explained variance for a small a number as possible of PCs. We need to find an optimum number of PCs. This can be a non-trivial task because we do not want to be removing information from our data set. To search for the optimum number of PCs we use the residuals (E-matrix) and full cross validation [7] to further check that we have a good model.

3.2.2 Scores: T

The Scores (T), structure of the PCA, is a summary of the original X-variables that describe how the different samples in X (objects) relate to each other.

Using the score vectors of T we can plot the score plots. The score plot is one of the most powerful tools offered by a PCA. In a score plot the score vectors are plotted against each other. We can plot e.g. scores of PC1 against scores of PC2. For visualization we can make 2- or 3-dimensional score plots. In the score plot objects are plotted, the score vectors have the coordinates for the objects in PC-space.

3.2.3 Loadings: P

The Loadings (P), structure part of the PCA, are the weights of the X-variables on the scores T. They reveal which X-variables (attributes) are responsible for patterns found in scores, T. This can be seen of the loading plot(s), where the variables are plotted, while comparing with the score plot(s).

In a loading plot the loading vectors are plotted against each other and can be seen as a map of the variables. Again 2- or 3-dimensional plots can be made.

3.2.4 Residuals: E

The Residuals (E-matrix), is the noise part of the PCA, a (N x M) large Matrix. E is not part of the model. It will be the part of X which is not explained by the model TP^T. That is why it should generally not be "large", that would mean we have removed much information, which furthermore will not give us a good PC model. Hopefully, it should be no more than

noise, of some sort (e.g. differences in score on same sample and variable between assessors).

The evaluation of E is always relative to the average object of X (in variable space). This average object can also be thought of as the first approximation to X, or the 0th PC (PC_0).

As we calculate more PCs, the residual variance will change. We start off with a E_0 with a residual variance of 100% and an explained variance of 0% (in the model). Actually, at this stage we do not have the TP^T term, yet. TP^T equals to 0. The other way around, when we have a full model, E equals to 0 and we have 100% explained variance (but all the noise). Throughout the calculation of PCs:

$$ExplainedVariance + ResidualVariance = 100\% \tag{3.4}$$

This is what we use to find the optimum number of PCs. We can plot the change in residual variance, as the number of PCs increase, and by human interpretation select the best number.

The size of E is expressed in terms of squared variance. This is by proper statistical fashion, because E is an error term. So, there will be summation of squares of the elements of the E-matrix. This can be done either along the rows of the E-matrix, which will give us object residuals, or down along columns, which will give us variable residuals. For object residuals we will get a sum $e^2_{(j)} = \sum_{i=1}^{N} e^2_{(i,j)}$ for each row, of the E-matrix.

For the Manhattan plot we are interested in the residual variable variances, or actually the explained variable variances which we calculate of the residual variable variance. Residual variable variances can be calculated similarly to residual object variances. As already said, summation of squares down along the columns of the E-matrix, as such $e^2_{(i)} = \sum_{j=1}^{M} e^2_{(i,j)}$ for each column. One summation for one column will hold the residual variable variance for a certain variable (attribute). We calculate the cumulative explained variable variance by $explained_{(i)} = 1 - norm(\sum_{j=1}^{M} e^2_{(i,j)})$. In the Manhattan plot we will see the cumulative explained variable variances plotted directly.

Now let us take a look at the total residual variance, for all the objects simultaneously. We get a measure of the distance between the objects in variable space and the representation of the objects in PC-space by this formula:

$$e^2_{tot} = \sum_{i=1}^{N} e^2_{(i)} \tag{3.5}$$

The smaller these distances are, the closer the PC model representation of the objects is to the original objects. This sum will be decreasing as the number of PCs increase ($e^2_{tot0} > e^2_{tot1}$).

3.3 PCA Criterion & Solution

We wish to find the directions that are orthogonal to each other and where there is largest variances (as explained earlier) of multivariate data (now let us call the data matrix A instead of X). We want to find the structure that forms the PC space. This is something we can call the PCA criterion. It has been proven that the eigenvectors of the covariance matrix of a matrix A point in the directions we wish to find (see [7] for theory and further references). So, of this criterion we have an eigenvalue-eigenvector problem.

$$Ax = \lambda x \tag{3.6}$$

Definition: An eigenvector for a matrix A is a non-zero vector x such that $Ax = \lambda x$ for a scalar λ. The scalar λ is an eigenvalue of A.

Finding the eigenvectors of the covariance matrix, of our matrix A, is a special case eigenvalue problem. The covariance matrix of A is AA^T (a square matrix). For our eigenvalue problem the equation 3.6 can be rewritten as such:

$$(AA^T)x = \lambda x \tag{3.7}$$

The solution is usage of an algorithm that finds the eigenvectors mentioned above. The NIPALS algorithm is one example for this purpose (see section 3.3.2). Finally the scores and loadings must be obtained of the results.

3.3.1 Singular Value Decomposition

The Singular Value Decomposition (SVD) has an expression that fits the eigenvalue problem of our PCA criterion. It can be used for a matrix A of any N x M (i.e. square or rectangular), it is closely related to PCA and is also often implemented in linear algebra software packages (e.g. in LAPACK, GNU Scientific Library, JAMA, R, SciPy and more).

The SVD factors a matrix A into USV^T, with singular values $\sigma_1 \geq ... \geq \sigma_{min(N,M)} > 0$.

$$A_{(NxM)} = U_{(NxN)} S_{(NxM)} V^T_{(MxM)} \tag{3.8}$$

Carrying out SVD consists of finding the eigenvalues and eigenvectors of AA^T and A^TA. The eigenvectors of A^TA make up each column of V and the eigenvectors of AA^T make up each column of U. The singular values in S are the square roots of the eigenvalues from AA^T or A^TA.

The singular values are the entries on the diagonal of S and will be arranged in a descending order. The singular values are always real numbers.

Each of the eigenvectors represents a principle component. PC1 is defined as the eigenvector with the highest corresponding eigenvalue. The higher the eigenvalue, the more variance is captured.

The ability of the SVD fulfils our criterion and scores, loadings and residuals can be retrieved of the results. Carrying out PCA on a data matrix A is the equivalent to SVD on the covariance matrix of A.

3.3.2 Computing Eigenvalues & Eigenvectors

The SVD must use an algorithm for finding the eigenvectors and generally it is not easy. Here is one method for a square matrix A (as in our eigenvalue problem):

Before computing the eigenvectors we will find the eigenvalues. First we rewrite equation 3.6 as $(A - \lambda I)x = 0 =$ zero vector. $(A - \lambda I)x = 0 =$ is equivalent to $Ax - \lambda x = 0$. We can find an eigenvector by solving $(A - \lambda I)x = 0$ when we have an eigenvalue.

If $(A - \lambda I)x = 0$ has a nonzero solution we have $det(A - \lambda I) = 0$. $det(A - \lambda I)$ equals a polynomial of order N. The eigenvalues are the N roots of this polynomial. It remains to solve $(A - \lambda I)x = 0$ for each eigenvalue, which will yield in each eigenvector. The problem with this method is when we get higher order polynomials (the method is usable on at least 2 x 2 and 3 x 3 matrices).

In this thesis I will not go deeper into theory of eigenvalue problems. Nevertheless, one algorithm for finding eigenvectors is presented in the following section. For further reading see references [1], [2] or [5].

NIPALS Algorithm

The NIPALS ("Nonlinear Iterative Partial Least Squares") algorithm is another method of many that exists for finding (estimating) eigenvectors. It was originally made for PCA. Scores, loadings and residuals can be retrieved directly and the algorithm is iterative. The first PC is calculated of the first iteration, the second PC of the second iteration and so on. First the largest eigenvalue is found and then smaller and smaller ones.

Here is an overview of the algorithm:

Variables used in the iterations:

- A: is a data matrix of the variable space.
- $NumberOfPCs$: The desired number of PCs. $NumberOfPCs >= 1$ and $NumberOfPCs <= min(objects, variables)$

- $E_{(0)}$: meancenter(A), The E-matrix for the zero-th PC (PC_0)
- t: vector is set to a column in A. Will be the scores for PC_i
- p: will be the loadings for PC_i
- *threshold*: (≈ 0) to do the convergence check towards zero.

The algorithm:

1. Set $E_{(0)}$ equal to mean centered A
2. Set t equal to an acceptable attribute vector of A (a non-zero vector)
3. For each i in range(1, $NumberOfPCs$):
   (a) Project A onto t to find the corresponding loading p
   $p = (E_{(i-1)}^T t)/(t^T t)$
   (b) Normalize loading vector p to length 1
   $p = p * (p^T p)^{-0.5}$
   (c) Project A onto p to find corresponding score vector t
   $t = (E_{(i-1)}^T p)/(p^T p)$
   (d) Check for convergence to zero. If difference between eigenvalues $\tau_{new} = (t^T t)$ and τ_{old} (from last round) is larger than $threshold * \tau_{new}$ return to step (a)
   (e) Remove the estimated PC component from $E_{(i-1)}$
   $E_{(i)} = E_{(i-1)} - (tp^T)$
   (f) Collect scores and loadings: current t and p (and current $E_{(i)}$ if E-matrices are needed)

3.4 Manhattan Plot

The methodology of the manhattan plot involves all currently covered in this chapter. Mathematically beyond PCA the retrieving of the manhattan data is quite simple. It is just the matter of normalizing residual variances and calculating the explained variances. In the algorithm for retrieving the manhattan data the only interesting part of the PCA is the resulting residuals (E) and from these residuals the calculated explained variable variances.

3.4.1 Algorithm

While using PCA by NIPALS one can limit the number of PCs that shall be calculated, resulting in faster computation if e.g. only 4 PCs have been selected for PCA of the cheese data set. This advantage is not available for PCA by SVD in the implementation of PCA used in PanelCheck 1.3.0. The implementation of Manhattan plot in PanelCheck also allow preprocessing (mean centering and standardization) of data.

First a look at the variables used:

- X: is a data matrix of the variable space.
- N: The number of variables of X
- M: The number of objects of X
- $NumberOfPCs$: The desired number of PCs. $NumberOfPCs >= 1$ and $NumberOfPCs <= min(N, M)$
- E_0: ($meancenter(X)$) The E-matrix for the zero-th PC (PC_0)
- $ResidualVariableVariances_0$: The residual variance for each variable of E_0.
- $E_{collection(i)}$: The E-matrices for each PC_i, $i \in [1, NumberOfPCs]$
- E: The E-matrix for the current PC
- $ResidualVariableVariances$: The residual variable variances for the current PC
- RVV_{norm}: The normalized residual variable variances for the current PC
- $ExplainedVariableVariances_{(i)}$: The explained variable variances for each PC, $i \in [1, NumberOfPCs]$

The Manhattan algorithm:

1. Get mean center of X
   $E_0 = meancenter(X)$
2. Calculate the residual variable variances of E_0
   $ResidualVariableVariances_{0(j)} = \sum_{i=1}^{N} E_{0(j,i)}^2$
3. Carry out PCA of X
   $Scores, Loadings, E_{collection} = PCA(X)$

4. Get all E-matrices of the PCA

5. For each i in range(1, NumberOfPCs):

   (a) Get the E-matrix for the current PC
   $E = E_{collection(i)}$

   (b) $ResidualVariableVariances_{(j)} = \sum_{k=1}^{N} E_{(j,k)}^2$

   (c) Normalize residual variable variances
   $RVV_{norm(j)} = ResidualVariableVariances_{(j)} / ResidualVariableVariances_{0(j)}$

   (d) Calculate the explained variable variances for the current PC
   $ExplainedVariableVariances_{(i)} = 1 - RVV_{norm}$

T and P of the PCA can be returned along with the calculated cumulative explained variable variances as results of this algorithm.

3.4.2 Interpretation of Plot

In figure 3.3 on the next page we can see a typical plot, and in this case we have 6 attributes and 6 PCs. The plot in this example was saved as an eps image file by using PanelCheck's Manhattan plot. The sensory data is of an experiment with four television sets and where score values have been given on various attributes by human vision. In figure 3.3 on the following page you can see the plot with examples of some of the attributes that were used in this experiment.

The plot is a quad mesh with coloration of each quad. The colors indicate the level of explained variance. In the PanelCheck software it has been chosen to use only gray levels. From black to white, where black represent 0% explained variance and white 100% explained variance. The plot will show the user the cumulative explained variance that accumulates towards 100%. This means that if you look at e.g. noise and PC3 in figure 3.3 you will see the level of explained variance for PC3 plus the level of explained variance for PC1 and PC2.

Vertically one level (one row) is levels of explained variance for a certain PC. Horizontally one level (one column) is levels of explained variance for one variable/attribute. In the figure 3.3 we can see how the PCs are arranged. The arrangement of the attributes depends on how they are arranged in the data set file.

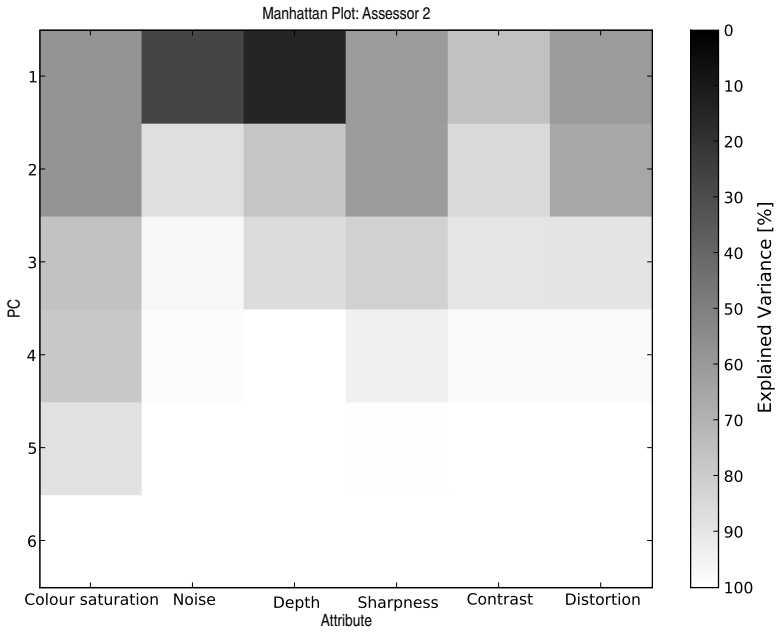

Figure 3.3: Manhattan plot example on a TV data set (courtesy by Bang & Olufsen).

Chapter 4

Software Tools

In the creation of PanelCheck different software tools have been used. The PanelCheck source code consists almost only of the programming language Python. So most of the packages used are made for Python. Common for all the packages used is that they are all Open-Source software. This chapter will cover these areas of the PanelCheck software and mention features, advantages and usage of the software tools used in PanelCheck. First let us take a look at the Python programming language.

4.1 Python

Python is a scripting language which is object oriented and based on the programming language C. Python can be compared to other scripting languages such as Tcl, Ruby or Perl. For one who has knowledge and experience at programming in at least one other programming language Python will be fairly easy to understand and learn. It is easier to read the code of Python compared to many other languages.

No matter how complex your program might get it will never be necessary to migrate to a higher level scripting set. The code becomes around 10-100 times more compact compared to that of C. About the same goes for the time it takes to write the code. This means that life should get easier for the programmer and that development goes faster. These are some of the reasons for why Python has been chosen as the programming language for PanelCheck.

Python is powerful and fast enough for a vast number of types of applications. It works well with other software technologies. It runs everywhere, at least on any platform there is a C compiler for. Python is under an open source license, it is free to use and distribute (also for commercial purposes).

The negative sides of choosing Python as the underlying "engine" for PanelCheck are such as possible performance issues and memory demand

(but not memory management!). But with Python even on this matter there are some solutions. One can adapt compiled code and use it in some of the more demanding areas of the program. It is something that is often used in the packages one can download for Python, and works great for e.g. numerical computing. We will come back to more on this later on, with a complete example in section 4.3 on page 38.

Background

Guido van Rossum began the work on Python at the National Research Institute for Mathematics and Computer Science in the Netherlands in the late 1980s. The initial goal for Python was that it would serve as a second language for people who were C or C++ programmers. To get the type of work done faster where writing a C program was inefficient. He wanted to bridge the gap between the shell and C.

It grew to be more than a scripting language. Many programs that are written in pure Python are not something you would call scripts, but rather applications or systems (PanelCheck would be an example of an application). Many developers now use it as their main language. The language has become very popular, for its clean syntax and rate of productivity. I would also say for part of its philosophy (as part of "The Zen of Python" [19]):

> *"There should be one « and preferably only one » obvious way to do it."*

I believe this type of thinking, for a programming language, aid in keeping focus under development and results in more maintainable software, especially for bigger projects. The projects will probably also be easier for more than one to cooperate on.

Documentation

There are several examples in Python in this chapter and in the following chapters. If you are new to Python, and want to learn about it (and programming), I suggest taking a look at the Python wiki online (`http://wiki.python.org/moin/`) and the beginner's guide there.

Yet a great deal of the code of some of the examples will probably be understandable. After all Python has a pseudo-code like style. I have more than once written very Python-like syntax for algorithms on paper, to get an overview, that I later wrote in C.

There is also a link to a list of books, for learning, on the online wiki (see reference [20]). Good documentation, something Python has, is important

and also helps accelerate the learning process. Documentation is available in various formats (such as HTML, PDF, Postscript and LaTeX), see `http://www.python.org/doc/`.

Other practical ways of getting information about anything in Python is by using pydoc. It can simply be used in a shell like this:

```
$ pydoc <name>
```

Where <*name*> is what you want information on, it can be e.g. modules, packages or functions. Here is another way of using pydoc shown in an interactive session (to start an interactive session just type *python* in a shell as also shown here):

```
$ python
...
>>>
>>> from pydoc import help
>>>
>>> import math
>>> help(math) # help(<name>)
```

Information about the in-built module math would now have been listed. The preceding example is also a Python code example on how to include modules and functions/instances of modules to your script.

In the interactive session as shown above you can enter and test Python code. More examples later on will be shown in interactive sessions. It is identified by the three "input" characters:

```
>>>
```

This also indicates that we are talking about pure Python code, which is usable in a script.

4.2 Python Packages

4.2.1 NumPy & SciPy

NumPy and SciPy are important packages for Python in the areas of numerical and scientific computing. NumPy makes Python fast enough for many and different tasks of number crunching.

Some History

NumPy is the successor of the Numeric package and is built on its array object. The Numeric package for Python was written mainly by Jim Hugunin

at MIT in 1995. Numeric had contributions of many other developers as well.

Travis E. Oliphant together with Eric Jones and Pearu Peterson started creation of SciPy in 2001, built for usage with Numeric, with many tools for scientific computing.

Due to limitations of the Numeric package a replacement was made by Perry Greenfield, Todd Miller, and Rick White at the Space Science Telescope Institute. This led to a split in the Python community of those who were developing for numerical computing, with some developing for numarray and others for Numeric and SciPy. Both had advantages and disadvantages compared to the other. E.g. numarray is fast for very large arrays (great for image processing), but slow for smaller arrays.

Early in 2005 Travis E. Oliphant began efforts of bringing the two sides together under one package for numerical computing in Python. He worked on the code of the Numeric package to make it more maintainable and flexible enough to implement important features of numarray. A hybrid array object was constructed. Later in 2006 NumPy 1.0 was released.

So, one can say there has been a bumpy road up to this point concerning Python and numerical computing. But NumPy is now the recommended package for numerical computing in Python. After the reunion and the birth of NumPy it is now in the opinion of many that NumPy should be included as part of the Python standard library. A course to take for pursuing this was decided at the SciPy conference in 2006.

During the work on PanelCheck this "issue" and change has been noticed. For the first official versions we used the Numeric package and SciPy. Usually large arrays (here meaning around 1000 x 1000 and larger) are not used of data sets loaded in PanelCheck. More importantly SciPy has many of the required methods. Later on we migrated to NumPy (and SciPy for this newer package). Due to the clean code of Python and good compatibility between the two packages, updating the code for NumPy was not problematic.

Features

NumPy The package has a powerful N-dimensional array object, efficient techniques for handling the array elements and methods for reshaping of arrays. The NumPy array can be interfaced with compiled code in a number of ways. This includes using f2py (for Fortran) and (for C) SWIG, cython, pyrex and by the NumPy C-API. NumPy include basic linear algebra functions, basic Fourier transforms and sophisticated random number capabilities.

SciPy The package is an Open-Source library with many scientific tools. It depends on NumPy and uses its array object.

The package includes modules for:

- statistics
- optimization
- numerical integration
- linear algebra
- Fourier transforms
- signal processing
- image processing
- genetic algorithms
- ODE solvers
- special functions

Efficiency Tips

Developing with NumPy is easy and, more importantly, can result in efficient code.

Whenever there is something typical you want to calculate of collection of data or you wish to manipulate an array in some way you should investigate if appropriate methods already exist in either NumPy or SciPy. This makes it easier, code may be more compact and very efficient. These methods are often interfaced to compiled code.

See the SciPy [24] or NumPy documentation [23] for finding the methods needed or try pydoc.

Array Creation & Indexing A NumPy array (ndarray instance) can be created of Python lists as shown below. Additionally in this case all the elements of the array are set to float types (if all elements of the *py_list* object had been int types we would have gotten an int array this way):

```
>>> from numpy import *
>>>
>>> py_list = [1, 0.5, -3, 1.333]
>>> a = array(py_list)
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([ 1.   ,  0.5  , -3.   ,  1.333])
```

You can also create a zeros- or ones-array of a certain shape. The shape is set by a Python tuple of integers. The integers are the lengths of each dimension and the number of integers is the number of dimensions of the array. Here is an example:

```
>>> a0 = zeros((10, 10, 10), float)
>>> a1 = ones((2, 10), float)
```

The zeros-array *a0* is now a 3-dimensional array with the shape 10 x 10 x 10. And the ones-array is 2-dimensional of shape 2 x 10.

Indexing of NumPy arrays is done with Python list style syntax. One thing to be aware of on this matter is that this style of indexing:

```
a[0,0,0]
```

is more efficient than

```
a[0][0][0]
```

When working with matrices one can use the NumPy matrix object which e.g. makes matrix multiplication easy. However, where possible you should use the array object instead for better performance. E.g. the *identity* method, another method for array creation, yields a square 2-dimensional array object for representing the identity matrix. But you can easily convert between the matrix object and array object, as shown here:

```
>>> I = identity(3, float)
>>> I
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> I = matrix(I)
>>> type(I)
<class 'numpy.core.defmatrix.matrix'>
>>> I = array(I)
>>> type(I)
<type 'numpy.ndarray'>
```

A thing that one must remember when working with 2-dimensional arrays for representing matrices (something usual) is correct indexing of rows and columns. The first index identify the row (or the y_order) the second index identify the column (or the x_order). You can also easily fetch whatever single row or column you wish as a 1-dimensional array:

```
>>> a = array([[1, 2],
...            [3, 4]])
>>> a[:, 1] # column 2
array([2, 4])
```

By using ":" we mean all the elements, as with usual Python list syntax. This technique also applies if we add more dimensions. For more dimensions the first index identify the outermost dimension, the second index one level in and so on. Here is an example of a 4-dimensional array with correct indexing:

```
a[w_order, z_order, y_order, x_order]
```

Array Iteration For iterating an array we can use plain Python for-loops:

```
>>> (rows, cols) = a.shape
>>> for row_ind in range(rows):
...     for col_ind in range(cols):
...         a[row_ind, col_ind] = a[row_ind, col_ind]**2
```

But they are slow, easily too slow when working with e.g matrices. For larger arrays one should use *xrange* (iterator) instead of *range* for conserving memory and better performance, but it is still far slower than such functions in compiled code. There is another technique we can try using NumPy features. The NumPy array support vectorized expressions and the two for-loops above can be replaced by one line:

```
>>> (rows, cols) = a.shape
>>> a[0:rows, 0:cols] = a[0:rows, 0:cols]**2   # or simply: a = a**2
```

This will yield in a greater performance gain. But in some cases this technique might be inconvenient (or impossible) to set up. Here is another "dummy" example showing usage of *xrange* and vectorized expressions:

```
>>> (rows, cols) = a.shape
>>> for row_ind in xrange(rows):
...     a[row_ind, 0:cols] += (1.0 + row_ind)
```

In section 4.3 an example on how to iterate NumPy arrays in a C extension module is presented. This along with results of performance using different methods of iterating an array.

4.2.2 Matplotlib

Matplotlib is a 2D plotting library for python written and maintained by John Hunter (with contributions by a number of other developers). It has a fully object oriented API and can utilize many graphics-toolkits as backends. Matplotlib can produce high quality plots suitable for publication, also including the possibility of mathtext. There are anti-grain rendering capabilities so the plots will look nice and smooth (reference [28]). One can also store plots as PNG, postscript (for inclusion in TEXdocuments) and other image file formats.

Pylab

The plot library also comes with an interface called pylab which is meant to mimic the ease of plotting such as in matlab. Here is an example Python script with usage of this interface:

```
#!/usr/bin/env python
from pylab import *

# horizontal axis values (time):
t = arange(0.0, 1.0+0.01, 0.01)

# vertical axis values (AC voltage):
s = cos(2*2*pi*t)

# plot with default settings:
plot(t, s)

# set label names:
xlabel('time (s)')
ylabel('voltage (mV)')
title('Nice and simple ;)')

grid(True) # set grid on:
savefig('simple_plot.eps') # save plot as image file
show() # create frame and show plot
```

Which would create and show a frame with the plot seen in figure 4.1 on the following page. The frame title would be "Figure X" as default matlab [26] style.

With use of the pylab interface matplotlib contributes to ease the creation of new functions and prototype modules. There can be faster testing of new visualizations of any 2-dimensional type of plot.

John Hunter started the development of matplotlib because he was straining with limitations of matlab as a programming language and wanted to move to using python instead. Not being able to find a 2-dimensional plotting library that met his requirements he started on a new plotting library for python.

I would guess that python + matplotlib + NumPy & SciPy would substitute (or even better than just substitute) matlab well for many matlab users. Some advantages would be: a free package, Python as the programming language (with the many advantages that brings), still beautiful plots and plenty of other packages and documentation available. Disadvantages would be, mainly: lack of functions (that might take much time to implement in Python), less software support, no warranties, possibly slower code and calculation.

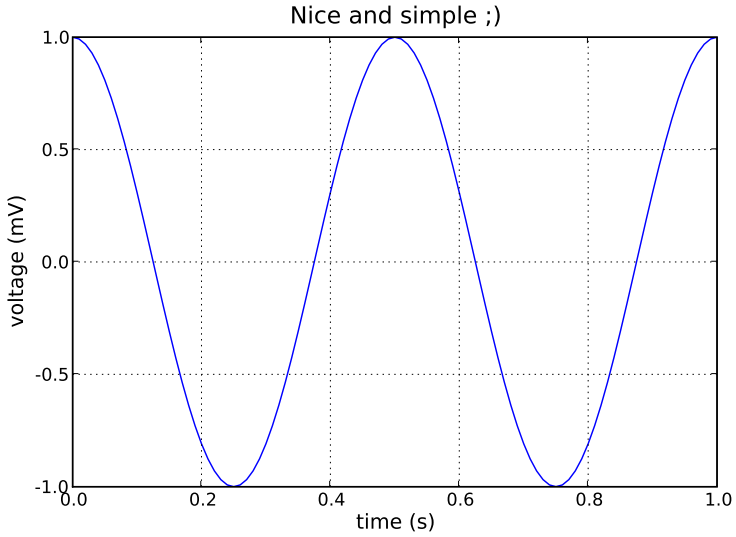

Figure 4.1: Simple plot using the pylab interface.

Object Oriented API

Two important classes concerning object oriented programming (OOP) in matplotlib are the Axes and Figure classes. These are the classes that control the plot. In matplotlib both the Figure and Axes classes are subclasses of the Artist class. Every class that is meant to be visualized as part of a plot is a subclass of Artist (e.g Polygons, Rectangles and more). The Artist class is an abstract class for rendering into a canvas and takes care of transformation of units. The matplotlib class library is available on this web page: `http://matplotlib.sourceforge.net/classdocs.html`

The Figure object can hold one or more Axes objects and controls positioning and presentation. Together with the axes it can also hold labels, legends, colorbars and other doodads you would want to add for making your figure informative and nice looking. Many such "doodads" are available in matplotlib.

The Figure and Axes classes are used for creating plots and can be used in this way:

```
>>> from matplotlib.figure import Figure
>>> fig = Figure(None)
>>> axes_rect = (0.1, 0.1, 0.8, 0.8)
>>> ax = fig.add_axes(axes_rect)
```

Now we have a matplotlib figure (variable: *fig*) with an empty plot with default settings. To be able to save it as an image or view it we need to

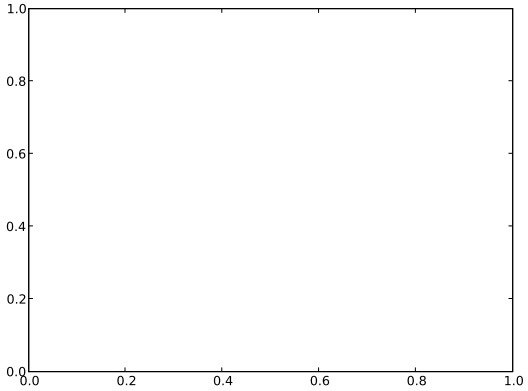

Figure 4.2: Test plot 1 (empty plot) using the object oriented API of matplotlib.

render it to a canvas. To do this we must use a backend (default is Tk, other good backends are e.g. wx or Qt). PanelCheck uses the wx backend. Here we will see this done with the standard backend and while using the anti-grain (agg) capabilities.

```
>>> from matplotlib.backends.backend_agg import FigureCanvasAgg
>>> canvas = FigureCanvasAgg(fig)
>>> canvas.print_figure("test.eps")
```

The print_figure command renders the plot and, in this example, writes it to disk with filename "test.eps" in the format given by the file extension. The result can be seen in figure 4.2.

A good practice with matplotlib is to stick with collections when you want to visualize e.g lines or polygons, for increased performance of the rendering. Collections can be created of lists/tuples/arrays (also NumPy arrays) of numbers (or points) and are added to Axes object. Here is an example where we add a line collection to the axes, use RGB tuples for coloring of each line and set new horizontal axis limits:

```
>>> from matplotlib.collections import LineCollection
>>>
>>> # list of RGB tuples with custom colors:
... # red, green, blue, yellow, magenta, cyan
... rgb_tuples = [(1, 0, 0), (0, 0.867, 0), (0, 0, 1), \
...               (1, 0.8, 0), (0.8, 0, 0.934), (0, 0.934, 0.734)]
>>> vertices = []
>>> for ind in range(len(rgb_tuples)):
```

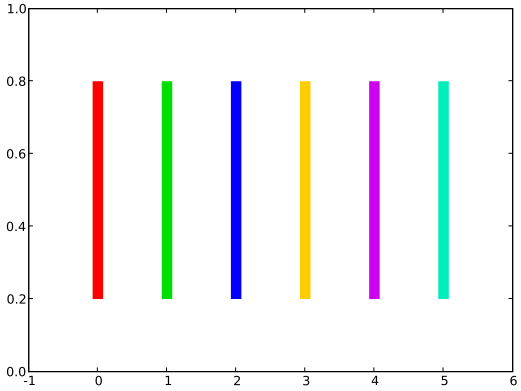

Figure 4.3: Test plot 2 (with a line collection) using the object oriented API of matplotlib.

```
...     # append one line: ((xmin, ymin), (xmax, ymax))
...     vertices.append(((ind, 0.2),(ind, 0.8)))
...
>>> lc = LineCollection(vertices, colors=rgb_tuples, linewidths=10.0)
>>> ax.add_collection(lc)
>>> ax.set_xlim(-1, len(rgb_tuples)) # xmin, xmax
(-1, 6)
>>> canvas.print_figure("test2.eps")
```

This will produce an image as seen in figure 4.3. The Axes object has many methods for adding elements for visualization. This plot and the plots created with pylab are affected by a number of default settings. The settings can be adjusted in the "matplotlibrc" file found under the mpl-data folder where matplotlib is installed.

The PlotData (see appendix B.2 on page 87) class of PanelCheck stores a Figure object and the current Axes object (along with many settings for the plot and calculation). When we make overview plots in PanelCheck the Figure object of PlotData will have more than one Axes objects.

Further reading, documentation, software and installation guide can be found at the project web site `http:\\matplotlib.sourceforge.net` [27].

4.2.3 wxPython

wxPython is the graphics user interface (GUI) toolkit used in PanelCheck. The toolkit is a wrapper package of wxWidgets which is an API for writing GUI applications. wxWidgets is written in C++ and is a platform independent GUI library. Python bindings are provided to this library with wxPython. As wxWidgets, wxPython is also a platform independent package.

The wxWidgets project was started in 1992 at the University of Edinburgh by Julian Smart. The project was started for creation of portable applications across Unix and Windows systems. Later on support for the Mac platform and other platforms was added as well. It will link to appropriate libraries on the system and will adopt the correct look and feel for that platform. Furthermore, it is a library that is easy to use, has great GUI functionality and it also provides a number of other features such as: network programming, streams, multi-threading, clipboard functionality, image handling (with support for many formats), database support, HTML viewing and much more. These capabilities are also available in wxPython.

The wxPython project was started by Robin Dunn. He was searching for an option for deploying GUI on Unix and Windows systems. When he came across Python bindings to the wxWidgets library the search was over. He learned Python and became one of the main developer of the wxPython toolkit. At first there was just a few "hand-made" bindings to the wxWidgets library, but this changed with the use of SWIG.

SWIG stands for Simplified Wrapper and Interface Generator and is a tool that allows developers to wrap C or C++ functions for use in scripting languages (Python is among the supported languages). By use of SWIG the amount of work in binding Python to wxWidgets decreased greatly. And after a while, in 1998, the "modern" wxPython was announced.

wxPython is a stable, robust, simple, easy-to-use GUI toolkit for Python. However it is not the standard GUI toolkit. Many who develop GUI applications in Python are surprised by this. In response to this here is a quote [29] of Guido van Rossum, the creator of Python:

> *"wxPython is the best and most mature cross-platform GUI toolkit, given a number of constraints. The only reason wxPython isn't the standard Python GUI toolkit is that Tkinter was there first."*

Application Example

Just to show how simple usage is, here is an example program where a frame is created with some content (I have also added comments in the code with explanations and some tips):

```
#!/usr/bin/env python
# Import the wxPython GUI package
import wx

# Create a new frame class, derived from the wxPython Frame.
class SubClassWXFrame(wx.Frame):
    def __init__(self, parent=None, id=-1, title="The Frame",
                       pos=wx.DefaultPosition,
                       size=wx.DefaultSize,
                       style=wx.DEFAULT_FRAME_STYLE):
        # initialize Frame (self)
        wx.Frame.__init__(self, parent=parent, id=id, title=title,
                                pos=pos, size=size, style=style)

        # create a panel to hold all GUI elemets (with self as parent)
        panel = wx.Panel(parent=self, id=-1)
        # Tip: wxPython Panel class is good to use as
        # SuperClass for creating MegaWidgets

        # create a text object
        label = wx.StaticText(parent=panel, id=-1,
                              label="Input field:")
        # Tip: setting id= -1 will automatically generate an
        # appropriate id (int) for the wx object

        # create a input text field with multiple lines
        self.text_field = wx.TextCtrl(parent=panel, id=-1,
                                      style=wx.TE_MULTILINE)
        # Tip: Here adding object to self for easy access of
        # its properties and methods

        # create a basic button with a text label
        button = wx.Button(parent=panel, id=-1 , label="Reset")

        # create a sizer for layout of gui elements vertically
        sizer = wx.BoxSizer(wx.VERTICAL)

        # add wx.Window objects to sizer for nice layout
        sizer.Add(label) # trusting default layout options
        sizer.Add(self.text_field,
                  1,         # make vertically stretchable
                  wx.EXPAND) # make horizontally stretchable
        sizer.Add(button,
                  0,               # make vertically unstretchable
                  wx.ALIGN_CENTER) # centre horizontally


        # event handling:
        # bind function to button, triggered by event
        # wx.EVT_BUTTON (button clicked)
        button.Bind(wx.EVT_BUTTON, self.onButtonClick)

        # bind function to frame, triggered by event
```

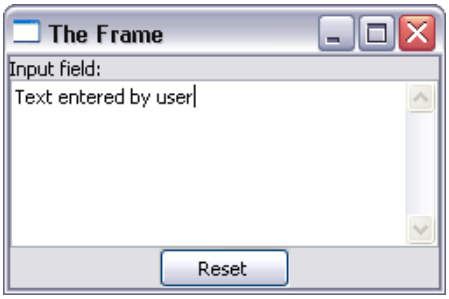

Figure 4.4: Example wxPython Frame.

```
54          # wx.EVT_CLOSE (when frame is closed)
55          self.Bind(wx.EVT_CLOSE, self.onCloseWindow)
56
57          # set sizer to wx.Panel
58          panel.SetSizer(sizer)
59
60      def onButtonClick(self, event):
61          # sets text_field to contain just empty string
62          self.text_field.SetValue("")
63
64      def onCloseWindow(self, event):
65          # destroys the window safely
66          self.Destroy()
67
68  # Create a new application class, derived from wx.App.
69  class Application(wx.App):
70      def OnInit(self):
71          # create custom frame class
72          frame = SubClassWXFrame()
73          self.SetTopWindow(frame)
74          frame.Show()
75          return True
76
77  if __name__ == "__main__":
78      app = Application()
79      app.MainLoop()
```

For further reading see reference [29]

4.2.4 Other Packages

Other important packages used in PanelCheck are pywin32 [32] and Rpy [34]. Here is a brief overview of the two packages:

The pywin32 package

The package enables communication with and use of windows extensions. This includes use of win32 API and Component Object Model (COM) support. COM is a platform for a software component system (for compatibility) introduced by Microsoft in 1993. It enables interprocess communication and dynamic object creation in any programming language that supports the COM technology. This means that through COM objects one can e.g. access and use other software installed on that windows. As e.g. PanelCheck uses COM objects for reading Excel files and exporting PowerPoint files. Remark: the newer .NET technology will be replacing COM (OLE, OLE Automation, ActiveX, COM+ and DCOM) to some extent. Python can be integrated with .NET. For .NET and Python IronPython is a good option. A side note on IronPython: it was actually started by Jim Hugunin, the creator of the Numeric package, he was later on hired by Microsoft to continue the work on Python for .NET.

Rpy

Rpy is a package which delivers a simple and robust interface to the R programming language. It can handle all R objects and all errors from the R language are converted to Python exceptions. Any module installed for the R system can also be used from within Python. R [34] is a free software environment for statistical computing and graphics. The R language is built on C and C++. It is an Open-Source project available on many platforms.

In the PanelCheck software the Rpy package is used to access R scripts from Python. The Danish "PanelCheck" team uses R to implement their statistical methods. The calculations and analysis are carried out by the R scripts, results are transferred back to Python and finally visualizations are made with matplotlib. Rpy enables Matforsk and their collaborative partners in Denmark to use each others statistical methods, even though the methods are written in different programming languages. PanelCheck comes with a stripped version of the R package.

4.3 C & Python

Python is built on C [15][16]. Representation of the Python objects in C is done by using C structs (as C is not an object oriented language). To create a C extension for Python we must use these structs and parse the incoming Python parameters to structs in the method of our C code. This special parsing method is important because in C there is strict typesetting, but there is no typesetting of instances in Python.

Compiled Module Example

To create a C extension for Python you must obviously have some experience with C and you should have the Python C-API [21] available. If the C extension involve using the NumPy array object you should also see the C-API section of the NumPy documentation [22].

Let us implement plain loops iterating a 2-dimensional NumPy array object with a simple calculation for each element (example 1):

```
>>> (rows, cols) = a.shape
>>> for row_ind in range(rows):
...     for col_ind in range(cols):
...         a[row_ind, col_ind] = a[row_ind, col_ind]**2
```

The improved version using NumPy features (example 2):

```
>>> (rows, cols) = a.shape
... a[0:rows, 0:cols] = a[0:rows, 0:cols]**2   # or simply: a = a**2
```

We can exchange the preceding code with the following code, for usage of the C extension module for Python (example 3):

```
>>> import _loop_test
>>> (rows, cols) = a.shape
>>> _loop_test.c_loop(a, rows, cols)
```

There are some requirements for the C extension. Firstly we must have the correct includes for access to Python C-API structs and methods.

```
// C extension for Python (source code: loop_test.c)
#include "Python.h"
#include "/usr/lib/python2.5/site-packages/numpy/core/include/numpy/arrayobject.h"
```

We must create a struct listing all the methods that are to be available in Python.

```
static PyMethodDef loop_test_methods[] = {
  {"c_loop", c_loop, METH_VARARGS},
  {NULL, NULL}};
```

A function must be added for initialization of the module.

```
PyMODINIT_FUNC init_loop_test()
{
  Py_InitModule("_loop_test", loop_test_methods);
  import_array();  // for usage of NumPy
}
```

Then the functions we want to be available in Python must be made. Those functions should be placed before creation of the methods table and the initialization method. Each of these functions could be declared like this (while only changing the function names):

```
static PyObject *c_loop(PyObject *self, PyObject *args)
{
  // parsing, safety checks, calculation, ..
}
```

The function must first parse the arguments. Here is an example on parsing of multiple parameters:

```
int rows, cols; // shape
PyArrayObject *X; // NumPy array

/* get arguments:  */
if (!PyArg_ParseTuple(args, "O!ii:c_loop", &PyArray_Type, &X, &rows, &cols))
{ return NULL; }
```

There should be some safety checking of the input parameters.

```
if (NULL == X) { return NULL; }

if (X->nd != 2){
  PyErr_Format(PyExc_ValueError,
  "array has wrong dimension (%d)",
  X->nd); return NULL;
}
```

Then set up pointers to each row of the NumPy array. This is a pretty general solution for a 2-dimensional NumPy array.

```
double *data_ptr; double **x; int i;
data_ptr = (double *) X->data; /* is a PyArrayObject* pointer */
// shape could also have been fetched here (of X):
// rows = X->dimensions[0]; cols = X->dimensions[1];
x = (double **) malloc((rows)*sizeof(double*));
for (i = 0; i < rows; i++)
{ x[i] = &(data_ptr[i*cols]); /* point row no. i in X->data */ }
```

After that the desired calculations can be carried out and an acceptable value must be returned.

```
int r; int c;
for (r = 0; r < rows; r++)
```

```
{
  for (c = 0; c < cols; c++)
  {
    x[r][c] = x[r][c] * x[r][c];
  }
}

free(x); // free any allocated memory
return PyInt_FromLong(1); // return value (acceptable Python object)
```

The basics of a creating a C extension using the C-API have now been shown with a "simple" functional example, including usage of a 2-dimensional NumPy array. It remains only to compile, link and test the module. This is done by using a C compiler and creating a shared library object. This is how it (_loop_test.so) could be built on Linux:

```
$ gcc -I /usr/include/python2.5/ -c loop_test.c
$ gcc -shared -o _loop_test.so loop_test.o
```

Here are results of the three code examples 1, 2 and 3 used with a matrix of size 3000 x 3000 of random values (I got similar performance differences using matrices of other sizes):

```
Example 1: 264.8   (using Python for-loops)
Example 2:   3.1   (using NumPy features)
Example 3:   1.0   (using C extension)
```

We get quite the leap in performance going from code A to B. For more advanced cases migration to C would probably be more important. To achieve best performance we must create an extension module in C (or Fortran), but usage of the efficient possibilities of NumPy might work well enough, as in this case I would say (considering the time it takes to create a C extension module).

Chapter 5

The PanelCheck Software

5.1 Development History

The first actual development of PanelCheck, on the software side, started as a student bachelor's degree project at University College of Oslo in cooperation with Matforsk in January 2005. We were three students including myself, together with dr. scient. Oliver Tomic in the planning and development of the software.

Oliver had much experience with Python and wanted it to be the language for PanelCheck. Even though we were informatics students this was still our first encounter with the Python language. This, however, did not prove to be a big issue in the development stage. Oliver was an important resource and a good teacher so learning went fast. To me, after a few weeks of testing and learning about Python and the Numeric package it was apparent that Python was a good choice for this software tool.

Design Goals The design goals for the prototype were as following.

- Efficient and fast usage
- Easy to use
- Nice and clean user interface
- Maintainable code that is possible to develop further

The design goals still stand for newer versions of PanelCheck.

The goals were met, to some degree, and the prototype was finished in May 2005 with two plot types implemented. More than two plot methods were planned for implementation, but still, the PanelCheck project members at Matforsk were pleased with the two methods of the prototype and

Figure 5.1: The PanelCheck logo.

the resulting software tool. After the student project was over I got the opportunity and pleasure to continue working on PanelCheck for Matforsk.

Development proceeded and many plot methods and abilities were implemented. Finally the first official version was released in May 2006. There have been a number of releases until version 1.3.0, all from small fixes to greater additions of new possibilities and new statistical methods. Test-versions of 1.3.0 are now being tested at Matforsk and by partners of the industry. Version 1.3.0 is scheduled to be released in the end of May 2008, with the Manhattan plot as one of the new highlight features.

5.2 PanelCheck Architecture

The work on the PanelCheck software has a iterative development style, where new abilities are planned for the next iteration. PanelCheck has gone through a number of iterations of improvements. Actually at one point it was considered to re-plan and redo most of the architecture and code of the software. But by gradually making it more module-based, giving PanelCheck a more object oriented design, a complete re-coding of the software was not necessary.

5.2.1 Module Overview

Here, the module overview is a plan of PanelCheck and meant to give an overview of the internal and external Python modules, and importing of modules. Rather than showing the design of the software using e.g. UML diagrams or such formal languages, here the design will be presented with an informal diagram showing the module overview. Design diagrams as in this case are, in my opinion, often easier to interpret.

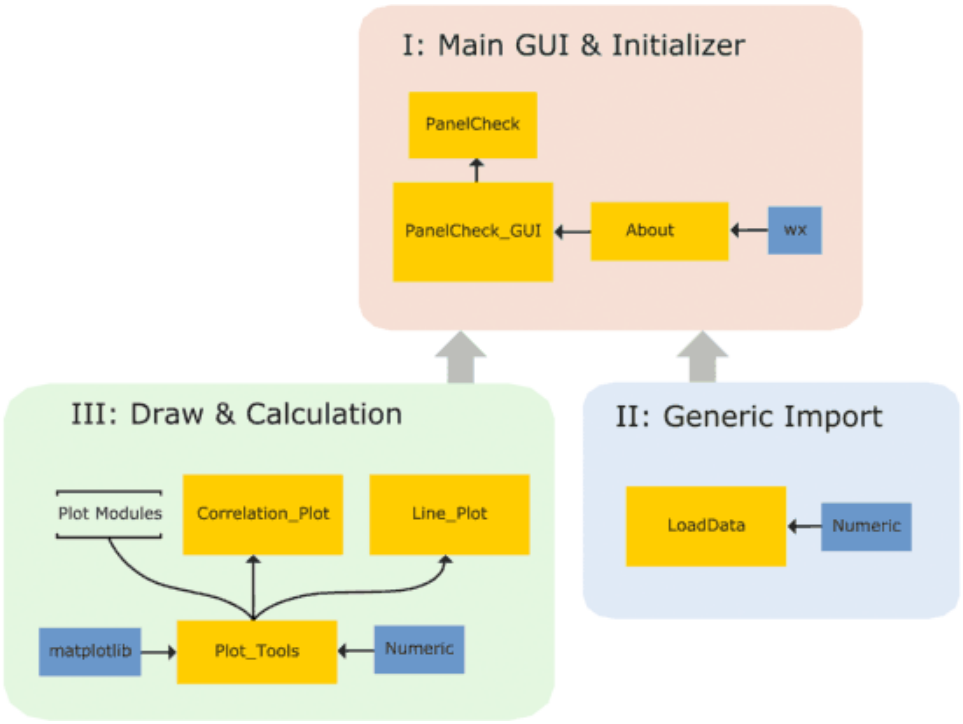

Figure 5.2: Module overview of the prototype version of PanelCheck

In figure 5.2 internal modules, being part of the source code of PanelCheck, are shown in yellow. External modules, the required installed ones on top of Python, are shown in blue. Arrows indicate importing of modules.

Prototype

Let us first take a look at the prototype module overview to see how PanelCheck first was built and better understand how it has evolved further. Figure 5.2 indicates that there are three main sections of the program. They were planned to be as following.

I: Main GUI & initiation This section has the main GUI and event-handling. All pointers and underlying objects are held and initiated through the PanelCheck_GUI module.

II: Generic import Generic import of data is handled in this section.

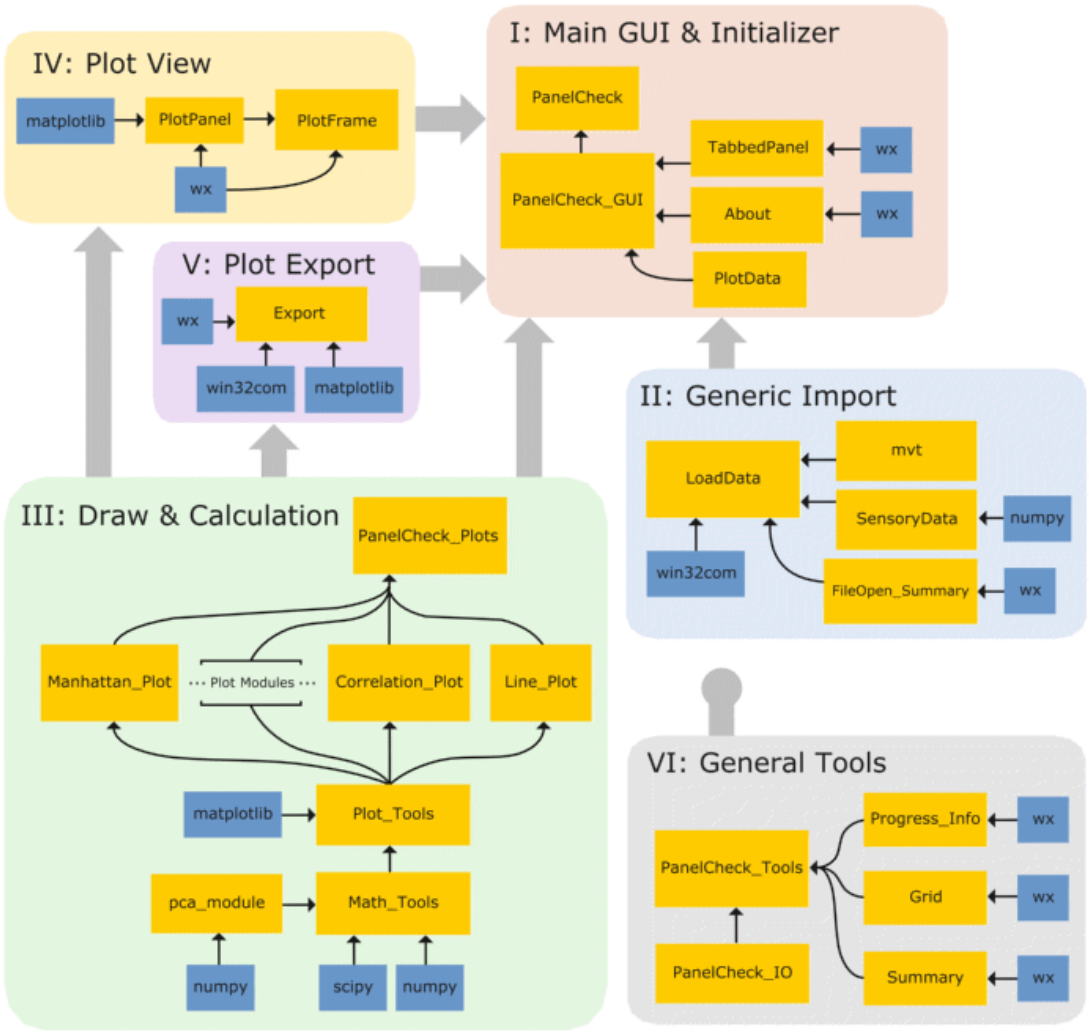

Figure 5.3: Module overview of PanelCheck version 1.3.0

III: Draw & calculation This section contains the modules for each statistical method and plotting of the visualizations. Creation of frame for viewing the plots is also handled in this section.

Version 1.3.0

The current version 1.3.0 still have the same three sections together with three new and a lot of additions to each of the first three, especially section III. There has also been some re-designing. The re-designing mostly involves taking the frame creation and viewing of plot out of section III and adding it as part of a new section. Additionally PanelCheck has been

made a lot more module based. This include using the instances/objects SensoryData and PlotData (see appendix B) for passing data. When these objects are used correctly each section of the module overview 5.3 should work, almost, independently. The Numeric package has been exchanged with NumPy. The code has also, of course, been adapted to the newest versions of the external modules.

I: Main GUI & main data store Initiates and constructs the main frame and user interface. The section has the main data store. All pointers and underlying objects are held and/or initiated through the PanelCheck_GUI module. The following modules are included in this section:

- **PanelCheck:** This module is the starter (has application main loop) and has command line argument handling functionality.
- **PanelCheck_GUI:** The module has the Main GUI, with event handling, and the main data store.
- **TabbedPanel:** Has standard class for gui elements in one tab and a subclass of this megawidget.
- **About:** Has frame for viewing a html file with credits and links.
- **PlotData:** Has standard class for passing plot settings and the plot itself (matplotlib Figure and Axes object).

II: Generic import The section handles loading of data from file and returns the SensoryData object for holding the data and file information. The following modules are included in this section:

- **LoadData:** Has the DataFile class with many control methods that are used during import. It has two subclasses PlainText and Excel for loading the data set from file on disk to a SensoryData object.
- **FileOpen_Summary:** Dialog for showing information and some user options while loading.
- **SensoryData:** Class for data store of the current data set. With information of the data set and many methods for retrieving selections of the data set as a NumPy array.
- **mvt:** The Missing Values Tools module has methods, for handling an array of NaN (Not a Number) values, which yield arrays with calculated valid number values instead of the NaN values.

III: Draw & calculation This section contains the modules for each statistical method and plotting of the visualizations. PlotData objects are returned. The plot tools of this section could here have been distinguished as an additional section.

Common for all the plot modules is that they take PlotData and SensoryData objects as input. Calculations are done based on settings of the PlotData object on selected data from the SensoryData object. The figure is created, plot is drawn and additional settings are applied. The PlotData object is returned.

The section has the following plot modules:
Line_Plot, **rawData_Plot**, **Correlation_Plot**, **Eggshell_Plot**, **profile_Plot**, **F_Plot**, **MSE_Plot**, **pmse_Plot**, **Tucker1_Plot**, **Manhattan_Plot**, **Consensus_Plot**, **MM_ANOVA_Plot**

The following modules are also included in this section:

- **PanelCheck_Plots:** Simply for inclusion of all plot modules for access of methods.
- **Plot_Tools:** Has general tools for all plot modules.
- **Math_Tools:** Includes math tools for all plot modules. Many selected methods of NumPy and SciPy, the pca module and a number of custom methods.
- **pca_module:** Has methods for carrying out PCA and some preprocessing methods and a method for retrieving correlation loadings. See section 6.2 on page 63.

IV: Plot view The section handles creation of a plot frame for viewing of the plot. The PlotFrame module handles interactivity in the plot and the possibility to go other plots. Additionally it has functionality for viewing of numerical data and raw data. The following modules are included in this section:

- **PlotFrame:** Has class PlotFrame, for viewing plots, which has buttons and options for interactivity, along with logic for handling next/prev plot. It utilizes most/all of the data of the PlotData object.
- **PlotPanel:** Has class for canvas (bitmap) for viewing of any PanelCheck plot. Also a class for custom drawing of information boxes available in all scatter/point plots, these boxes uses wx.DC (Device Context) for drawing and does not get included to the matplotlib figure.

V: Plot export This section takes care of exporting of images or PowerPoint files. No viewing of plots. The following modules are included in this section:

- **Export:** Has dialog with choices of exporting and methods for export of each plot type. One can export images (png or eps formats) or export a PowerPoint file.

VI: General tools The general tools section has methods and dialogs available in most other sections of PanelCheck. The following modules are included in this section:

- **PanelCheck_Tools:** Has some generic methods for use in various areas of the program. The following modules are imported.
- **PanelCheck_IO:** Handles customized binary I/O in PanelCheck, currently only for session data and the recent-files list.
- **Summary:** Has dialog class for viewing a scrollable text field.
- **Progress_Info:** Has a class for a small dialog with progress gauge and a text field for information. This is used in various areas of the program.
- **Grid:** Has grid frame class for viewing an Excel-like data grid with copy-to-clipboard functionality.

5.2.2 Data Flow

The data flow diagram in figure 5.4 on the following page gives a general overview of data flow for these three important functionalities of PanelCheck: Import, Plot and Export. In the figure you can for example see the input and output for a plot module's main function. You can see data transfer with usage of the SensoryData and PlotData objects.

There are a few things that are not covered in the diagram. One thing is that for some of the plotting methods there are optimizations where the calculated data is stored in the main data store and re-used for the next or previous plot if suitable. There might also be re-use of calculated data if a plot of the same type is created and the settings/selections have not been changed.

Another area not covered is about the PlotFrame class and its functionalities. This subclass of wx.Frame works almost as its own independent application, once created, with a set of functions such as next and previous plot. A list is kept in main data store of all the primary PlotFrames that are created so these PlotFrame objects and any of their children can be terminated.

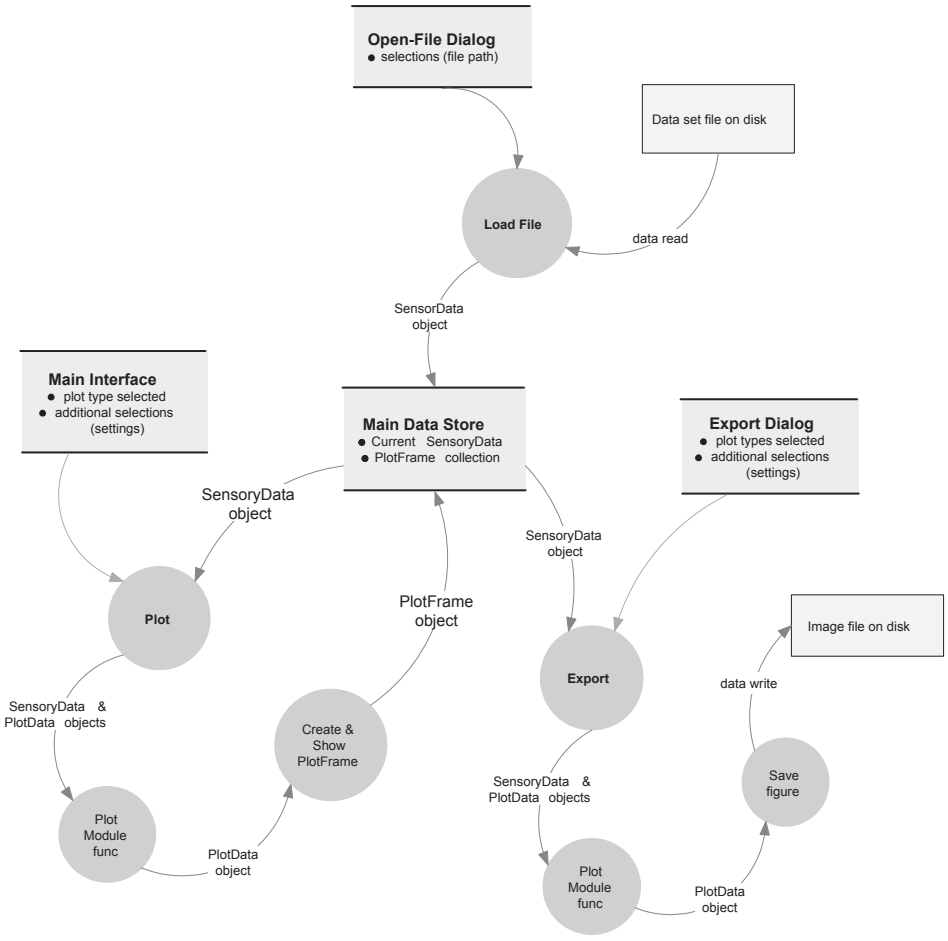

Figure 5.4: PanelCheck data flow diagram.

Two important objects for passing data in PanelCheck are the SensoryData and PlotData classes. Main parts of the source code, seen in appendix B on page 85, show details on these classes (code is taken from PanelCheck 1.3.0 test-version).

5.2.3 Data Structures

Data structure is the way of organizing and storing data for easy and efficient access. The structures are often grouped into two sections: linear data structures and data structures based on graph theory. PanelCheck makes use of standard Python linear data structures. Linear data structures are such as lists, tuples, hash tables (basically dictionaries in Python), arrays, stacks, queues and more. All these types of data structures are available in Python and can easily be utilized. In Python most of these data structures can also be nested in various ways.

The NumPy array is an important class which is widely used in PanelCheck. Especially for numerical computing the NumPy array can be very efficient and it is also fast for large arrays, even of multiple dimensions. There is more information about the NumPy package and array in section 4.2.1 on page 26 also showing efficient techniques of using the NumPy array.

There is no implementation of data structures based on graph theory in the PanelCheck source code (i.e tree structures, heaps and more). However, there is usage of nested linear data structures. This is something that is used in the SensoryData class. This class holds the data set to be analyzed. Other than that it also has information about the data file. Names of attributes, assessors, samples and replicates are held as lists of strings. Additionally there are several methods for fetching the numerical score values. Most of these methods returns a NumPy array object ready for number crunching.

When importing a file (data set) in PanelCheck a SensoryData object is created and "filled" with data. The SparseMatrix, seen in the SensoryData class in appendix B on page 85, is the main dictionary for storing the score values. Each row of scores are stored in a 1-dimensional NumPy array and each row is identified by a unique key. The key is a tuple of assessor name, sample name and replicate name. The dictionary can be used as shown here (assuming that you already have SensoryData object of a loaded data set):

```
key = ("Assessor 1", "Sample 1", "Replicate 1")

score_row = sensory_data_obj.SparseMatrix(key)

score_of_variable1 = score_row[0]
```

Data will mostly be retrieved by using one of the methods of the SensoryData class. But this is important to know if you were to create your own method for fetching data, in the SensoryData class, and shows an example of nested linear data structures in Python.

5.3 Implementing a Plot Module in PanelCheck

Before the implementation of a plot module in PanelCheck can begin development of the statistical method should be complete. However, experimental work on visualizations and methods goes fast by using Python and e.g matplotlib's pylab. This interface can also be used in PanelCheck, but not for the version of which an installer will be built.

For a possible future research version of PanelCheck, where experimental work would be implemented, addition of another plot module must be an easy and fast task. But for version 1.3.0 there are a few steps involved for doing this. I will refer to the sections and existing modules seen in figure 5.3 on page 45 and version 1.3.0 concerning details on implementation.

Three of the sections of figure 5.3 will be affected:

- **I: Main Frame** (standard GUI additions)
- **III: Draw & Calculation** (with a new plot module and possibly new math tools)
- **IV: Plot View** (most of the functionality here will work when using the PlotData class)

You can start the work on either section I or III. For section I: Main Frame you will have to add a standard tab to the GUI and add a function for building a custom tree (on which the user will double-click on an item to create a plot). For section III: Draw & Calculation you are a bit more free on how you wish to implement your plot module, but the main method must take SensoryData and PlotData as parameters and return either a "filled" PlotData object or None type.

I: Main Frame

The first thing to keep in mind is that in PanelCheck wxPython is used for all GUI elements (so have wxPython documentation [29] available). PanelCheck has a megawidget called TabPanel of the TabbedPanel module for a standard tab, and one subclass type of this megawidget. If additional GUI elements must be added (e.g more options for the plot) a subclass of TabPanel should be made.

Inclusion of a new tab to one of the four wx.Notebook parents (Univariate, Multivariate, Consensus and Overall) is done by creating a new TabPanel and passing the appropriate parent parameter. The new tab is created and stored in the MainFrame class of the PanelCheck_GUI module:

```
self.newplot_panel = TabPanel(parent=self.uni, func=self.create_plot)
```

One parameter must be the callback method *create_plot* of MainFrame for double-click mouse events of the TabPanel tree. Additional parameters can also be given for more settings on the new tab (see source code [13]). The *create_plot* method must be updated for handling data of the tree of the new tab and usage of the new plot module.

A custom tree must be built for the new tab and this must be done by a method available in MainFrame (see documentation on wx.TreeCtrl).

Finally, in a method called *update_tabs* of MainFrame the new tab object (self.newplot_panel) and the tree creation method must be added to appropriate lists there.

III: Draw & Calculation

There must be one main method that takes PlotData and SensoryData as parameters, and returns the PlotData object. This is common for all the plot modules.

In each plot module the Plot_Tools module (of section III in figure 5.3 on page 45) is imported. This tools module has a number of general methods that is used in the plot modules. Plot_Tools also makes the Math_Tools methods available to the plot modules. The Math_Tools module includes many NumPy, SciPy and pca_module methods and also has a number of custom methods.

A typical design and flow of a plot module in PanelCheck is as following:

- Error-checking on selections (additional error-checking later on is mainly meant for debugging of code)
- Get the required sensory data
- Statistical method calculations
- Drawing of plot on an axes
- Store numerical data for later viewing
- Store used raw data for later viewing

For fetching the required sensory data an appropriate method might exist in the SensoryData class, if not, a new one must be created there.

For aid on how to carry out the calculations on arrays and available methods see documentation of NumPy [22] and SciPy [24]. PanelCheck 1.3.0 uses matplotlib for creation of plots, see section 4.2.2 on page 30 for information on usage and references.

This is a standard code part used in most of the plot modules for creation of the Figure and Axes objects:

```
replot = False; subplot = plot_data.overview_plot
if plot_data.fig != None: replot = True
else: plot_data.fig = Figure(None)
if subplot: # is subplot
    plot_data.ax = plot_data.fig.add_subplot(num[0], num[1], num[2], aspect=plot_data.aspect)
else: plot_data.ax = axes_create(0, plot_data.fig, aspect=plot_data.aspect)
ax = plot_data.ax; fig = plot_data.fig # current axes and figure
```

The PlotData object (*plot_data*) is used here and the *num* list is passed as parameter to the main plot method if the plot is to have multiple Axes objects (an "overview" plot). The preceding code part makes the plot module compatible with the standard overview plot function of the Plot_Tools module. The *axes_create* method is another general method of the Plot_Tools module.

Numerical data and raw data must be stored in the PlotData object for later viewing. These data are viewed through an Excel-like grid and the values are store as list of lists (of strings or number values). Each element of the 2-dimensional list structure will be one element of a grid frame. This 2-dimensional structure does not have to be rectangular. The tools module has one method that sets the raw data for the given selections.

Finally the functions of the new plot module must be imported in the PanelCheck_Plots module.

IV: Plot View

Modifications/additions in this section is strictly not necessary, but for the prev/next buttons to be functional two methods must be updated, namely PlotFrame.get_tree_path and PlotFrame.replot. In get_tree_path the circle for prev/next is represented with a list and it must contain the tree data similar to that of the tree of the tab panel. This list might be a custom list of elements. The current tree data is available through the PlotData object and is the reference to navigate back and forwards in the list. Then the replot function can be updated so that it utilizes the new plot module.

Additional functionality one would want to apply to the plot frame for the new plot is more interactivity. For improved or desired interactivity in the plot customized code will have to be written. One is open to use possibilities of wxPython or matplotlib.

Summary

A general overview on how to implement a new plot module in PanelCheck that points to key aspects and some details for getting started.

Additional tips for creation of a plot module:

- See matplotlib examples on usage: `http://matplotlib.sourceforge.net/matplotlib_examples_0.91.2svn.zip`.
- Use general methods of the Plot_Tools module.
- Plot_Tools has a set of 14 colors, as a list of RGB tuples and HEX notation (used in many other plots).
- Create separated calculation methods and add unit-testing of these methods.
- See how things are solved in the other plot modules.

5.4 Quick-Guide on Using PanelCheck

There is more aid to get through *Help->Manual* with links to further reading and additional information. There are also some links through *Help->About*. The version number can also be seen in the about frame and also as part of the program title in the main frame.

5.4.1 Importing

After starting PanelCheck you will see empty GUI elements in each tab. These elements will be filled after we successfully load a data set. To import a data set simply select e.g *File->Import->Plain Text*, choose correct file type (depending on how your data set is stored), as shown in figure 5.5 on the next page. A standard file selection dialog will appear. You can also import a file through the console (e.g *Start->Run...* on Windows):

```
$ PanelCheck <data set file path>
```

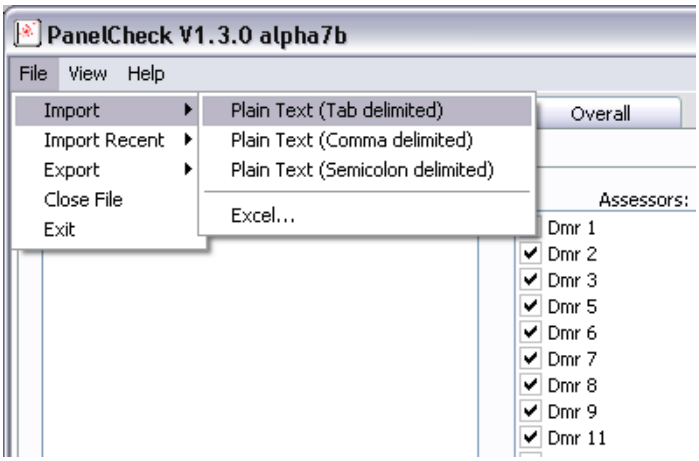

Figure 5.5: PanelCheck file importing.

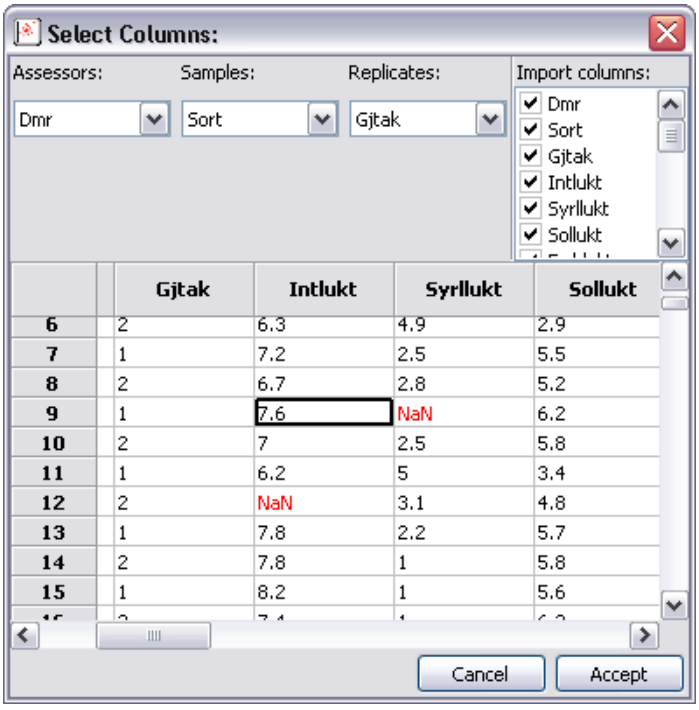

Figure 5.6: PanelCheck data import part I.

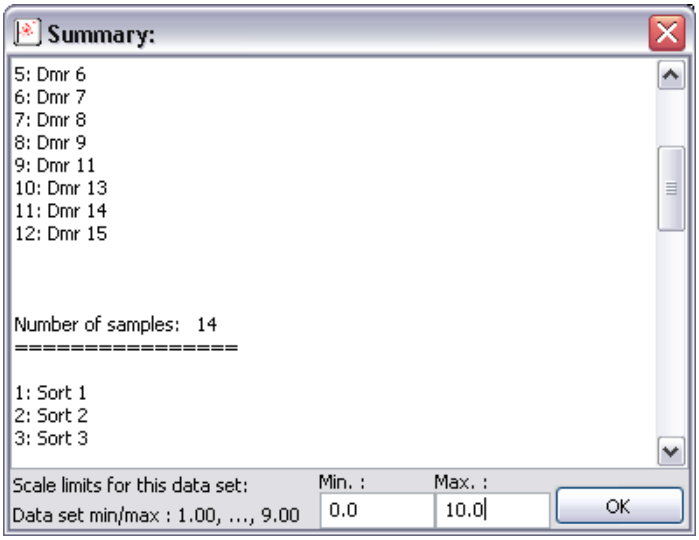

Figure 5.7: PanelCheck data import part II.

When you have selected the correct file you will have to go through two dialog windows for loading the data set. The first is seen in figure 5.6 on the preceding page. This dialog window shows options available before loading of the data set. You can check values of the data set. Missing values are shown in red as NaN values. You can select the correct columns for assessors, samples and replicates and you can also deselect any column you do not wish to be imported as part of the data set. After you have made correct selections and controlled the data set you press Accept to start loading.

The second and last loading dialog shows general information about the data set, see dialog shown in figure 5.7. In the lower section of the window you can see the real upper and lower limits of the score values of the data set. Here you can also set the scale limits used for many of the available plots. These limits should include the real limits of the data set. Afterwards you press OK and you will see the lists of the tabs be filled and a tree be built.

5.4.2 Plotting (Using Manhattan)

The primary tabs are divided into sections sorting the plot methods available. There are four primary sections: Univariate, Multivariate, Consensus and Overall. First you choose the primary tab and then the secondary tab which is the plot type you wish to create. There are unique trees for each tab, each plot method. After that you make selections on which assessors,

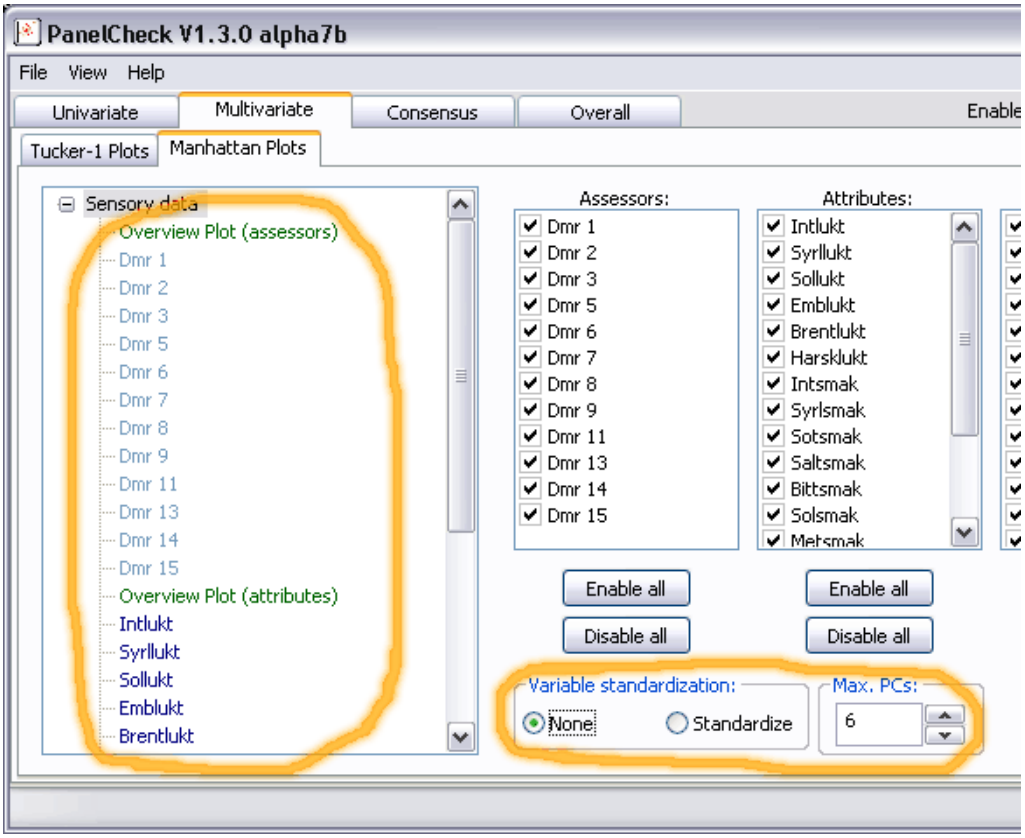

Figure 5.8: Manhattan Plots tab.

attributes and samples that are to be included in the calculations for the plot. To create a plot you must double-click on an item in a tree that is greenish or blueish (see figure 5.8).

When you double-click calculation for that method begins and the plot is created. At this point information dialogs concerning the calculation might be shown before the plot. The plot will appear in a plot frame that is similar for each plot. The plot frame has various buttons/options for the plot such as printing, copy-to-clipboard, save figure and more, as seen in figure 5.9 on the following page. You can hold the mouse over a button or icon to get more information about it with tool-tips. You can also easily and conveniently go to next or previous plot using the respective buttons visible in the lower-right section of the plot frame.

The preceding information is general for all the plot types. To create a Manhattan plot you must first click on the Multivariate tab and then on the Manhattan Plots tab. In the lower section of the tab you will see options

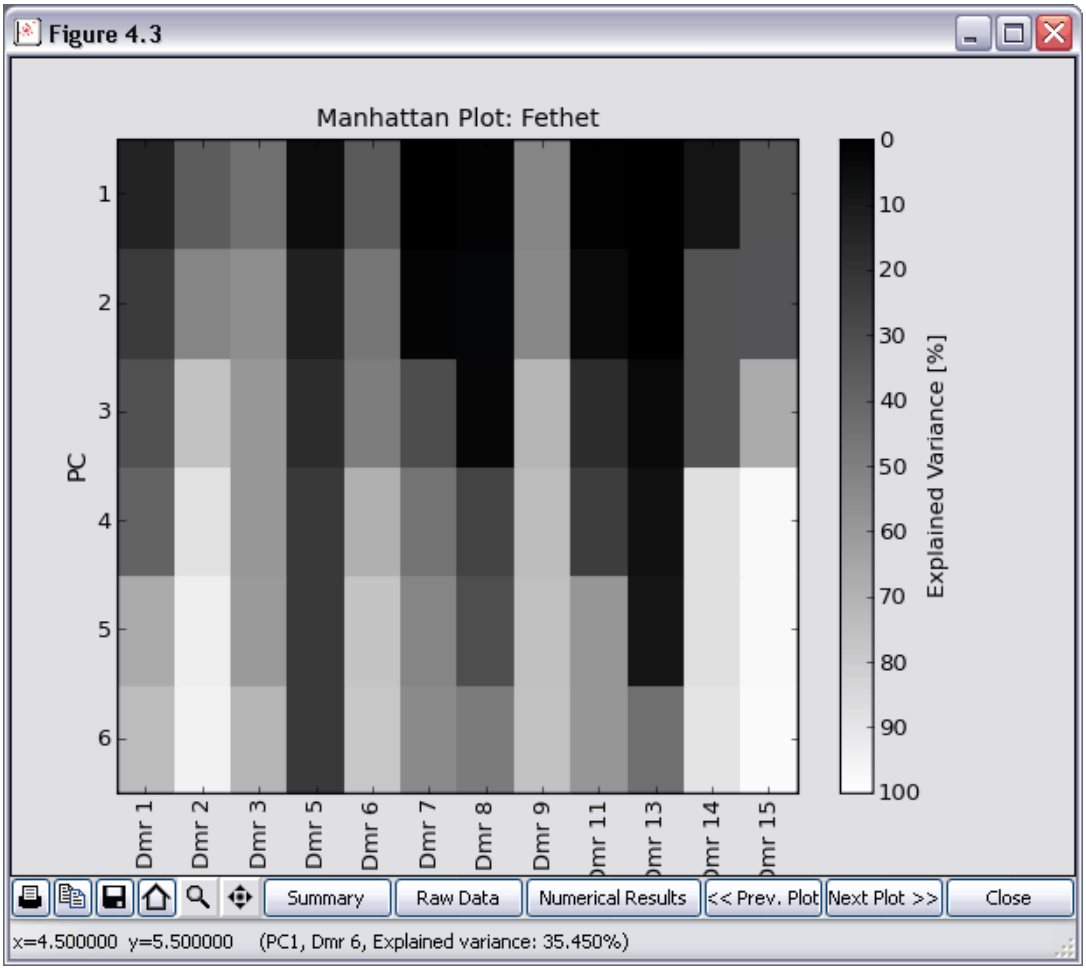

Figure 5.9: PanelCheck plot frame.

for a Manhattan plot (marked on figure 5.8). The options include desired preprocessing method and the number of PCs for visualization. After selections have been made you must double-click on an item in the tree to create a Manhattan plot. You will see Overview Plot (assessors), individual assessors, Overview Plot (attributes) and each attribute. Manhattan plot creation follow the easy-to-use design of PanelCheck. For example if you want to see the results for assessor 1, you double-click on the name of assessor 1 in the tree. If you double-click on the Overview Plot (assessor) item, the Manhattan plots for each assessor will be shown in one figure. And by double-clicking on the Overview Plot (attributes) item, the Manhattan data for each assessor will be shown with focus on each attribute instead. In the overview of many plots you can click on one plot to open it in a standard plot frame.

After creation of a Manhattan plot you can use the standard functions of the plot frame. You can e.g. see numerical results and raw data used by clicking on the respective buttons. You can also view the exact explained variance for a PC and an attribute by holding the mouse pointer over a quad, in the plot (information will be shown in the status bar). If you click on a quad an information box will pop up where you clicked with the exact explained variance for that quad.

5.4.3 Exporting

Another important feature available is exporting of plot to image files or to a PowerPoint file. To export files select e.g *File->Export->Image Files...* and the export dialog for images will be opened. You can select the plot types you wish to export and selections on active assessors, attributes and samples for each plot. Other options are path selection for storing the images and selections for image type and resolution (in dpi: dots per inch). For exporting to a PowerPoint file PowerPoint needs to be installed.

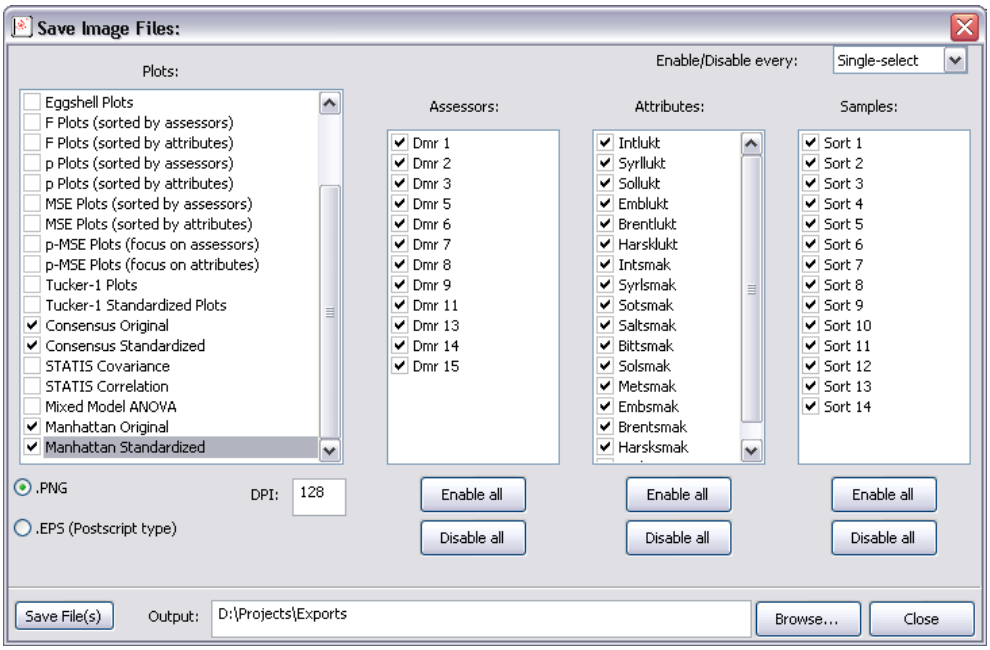

Figure 5.10: PanelCheck export dialog.

Chapter 6

The Manhattan Plot Module

A tool for visualizing multivariate data. Manhattan plot is basically an alternative way for presentation of the explained variances of a PCA. It was proposed mainly by Thobias Dahl, officially, in 2007. See article [3] in the references.

6.1 Design

The Manhattan plot module follow the standard plot module design of PanelCheck. For version 1.3.0 each plot module consist of a number of methods.

6.1.1 Algorithm

Before usage of the plot module there is a check on changes in the selections. The new selections are compared with active lists of the last PlotData object (see method in appendix B.2 on page 87 at line 32). If there are no changes the calculation collection of the last PlotData object is passed with the new PlotData object. In Manhattan plot a subclass of PlotData is used, it is called CollectionCalcPlotData and is found in the PlotData module.

Here is the general algorithm of what is done in the Manhattan module.

1. Check acceptable PlotData. No plot can be produced if there is empty selections of active assessors, samples or attributes. In case of empty selections an error message appear. Manhattan plot requires at least one active element for each list (at least two active samples). If error return None.

2. Decide type of projection and active assessor or attribute by information in tree data of PlotData.

3. If collection data is already calculated:
    (a) Get the calculated collection data of PlotData.
4. Else:
    (a) For each active assessor:
        i. Get active data X for current assessor. The method used can be seen in appendix B.1 on page 87 at line 43.
        ii. Check if any attributes should be left out and then update X. An attribute vector is left out if the STD = 0.
        iii. Collect information of which attributes that are left out.
        iv. Do calculation of Manhattan data for X (algorithm can be seen in section 3.4 on page 20).
        v. Collect Manhattan data for current assessor.
    (b) If any attributes were left out show information dialog about this.
    (c) Store collection of calculated data in PlotData object.
5. Do standard setup of Figure and Axes objects, for creation of plot.
6. Get plot matrix data, a 2-dimensional array of the explained variances for the active assessor or attribute, of the calculated collection data.
7. Create a polygon (quad) collection by plot matrix data with a given colormap.
8. Create point & labels list for information-labels in plot by clicking with mouse.
9. Add polygon collection to Axes object.
10. Get numerical data of calculated data and raw data of SensoryData object and store it in PlotData object.
11. Set adjustments of Axes and Figure (such as labels, title, limits, colorbar and more).
12. Set additional settings for PlotData object (plot type, lists, point & labels type).
13. Return PlotData.

In addition to the preceding functionalities the Manhattan plot module has two methods for overview plotting. One overview plot method for each projection type. Both of the methods makes use of a general overview method that is available through the Plot_Tools module. This general method is used in each case of overview plotting in PanelCheck.

6.1.2 Projections

There are two useful ways of projecting the results. Firstly, showing each attribute for one assessor. In this way you can see which attributes that are poorly explained for that single assessor. It is used for investigation on the individual assessors. Secondly, one can plot the results of each assessor with focus on only one attribute. The same data is used, but now you can see which assessors that have poor explained variance, compare agreement and structure for one single attribute. The projection types might also become even more useful by creating overview plots.

Both projections types have been implemented. There is a general method that creates the polygon collection and takes a 2-dimensional array (plot matrix data) of any size and of values from 0.0 to 1.0, as parameter. Another method fetches the correct plot matrix data from the calculated collection data depending on projection type and the active assessor or attribute. The calculated collection data is a dictionary where the keys are the names for each active assessor. For each active assessor we have a list of three NumPy arrays; scores, loadings and cumulative explained variable variances.

6.1.3 Testing

PanelCheck's Manhattan calculations can be tested by unit-tests using the Python script *testingPanelCheck.py* (see source code [13]). In the source code the folder *testing* contains data sets and the corresponding results for Manhattan calculations with different settings. Currently only parts of the Manhattan plot module is tested in this testing script, but the setup can easily be extended with testing of other plot modules of PanelCheck as well.

The meaning of unit-tests is to isolate different parts of the software and show that they work properly. Unit-testing is especially useful if you are modifying the code as a fast check of correctness. This is one of various benefits by using unit-testing [36].

6.2 PCA Module

This is a module for Python which depends on either NumPy or Numeric. The module carries out PCA by NumPy's SVD or by using my implementation of the NIPALS algorithm (see section 3.3.2 on page 19). I have also implemented the NIPALS algorithm in C as a Python extension. The E-matrices can be available immediately after the NIPALS algorithm has been carried out. The PCA module is used for the Manhattan calculations and comes with PanelCheck as part of the source code. It is also used in the other PCA-based methods of PanelCheck. To read more on the methodology of PCA see 3.2 on page 14. The PCA module is available as a separate

software tool [14].

Four different PCA methods have been implemented:

- NumPy-SVD PCA using NumPy array objects
- Python-NIPALS PCA using NumPy matrix objects
- Python-NIPALS PCA using NumPy array objects
- C-NIPALS PCA using NumPy array objects

Additionally methods for mean centering and standardization have been implemented.

6.2.1 Installation

There is a standard *distutils* [18] build and install:

```
$ python setup.py install
```

There are two variables that can be adjusted in *setup.py*. The first "add_ext" sets extension to be compiled and included (include: add_ext = True). If you set it to False, C Python extension will not be included and you cannot access c_nipals. Nevertheless, PCA can be calculated without the C Python extension. The other variable "old_numeric" sets which version to use. Either the Numeric version or the NumPy version (use NumPy version: old_numeric = False).

The Numeric version is more limited when it comes to functions. If possible the PCA module for NumPy should be used.

To test that installation did not fail, try:

```
$ python
...
>>> import pca_module
>>>
```

No errors or exceptions should appear.

6.2.2 Usage

Assume you have a 2-dimensional data matrix X that holds the (multivariate) data you want to analyze. This matrix is of size N x M, where N = number of objects and M = number of variables. Each row holds the values

of an object and each column holds the value for a variable. The data is M-dimensional in variable space.

Examples usage 1 (with total explained variance for each PC):

```
>>> from pca_module import *
>>> T, P, explained_var = PCA_svd(X, standardize=True)
>>>
```

Now both mean centering (always done) and standardization (standardize=True) of X has been done before the PCA. A matrix of the scores, a matrix of the loadings and an array of the total explained variances is returned.

Examples usage 2 (with the E-matrix for each PC):

```
>>> from pca_module import *
>>> T, P, E = PCA_nipals2(X, standardize=True, E_matrices=True)
>>>
```

There are two changes for the second example. Firstly, that the NIPALS algorithm is used instead of NumPy's SVD for PCA. Secondly, that we get the complete collection of all the residuals. The E-matrix for each of the first 10 PCs (if we do not specify the number of PCs).

Testing of the PCA module is also done by unit-tests with a Python testing script called *testing.py* which comes with the source code. There is also a method for time measurements in the testing script.

Running the testing script:

```
$ python testing.py
```

Or for the Numeric install:

```
$ python testing_numeric.py
```

errors - can be problems with the module (e.g. import error)
failures - occur if functions return wrong results

Chapter 7

Results & Discussion

7.1 Visualization

Colors

The explained variances are always in the range from 0% to 100%, from black to white on the gray-scale, respectively. On this matter the plot should be simple to deal with and it looks quite nice.

In the implementation I chose the coloring of the polygon collection to be by a colormap. This means there can be interesting (maybe more informative) variations of colors in a mapping 0.0 to 1.0. The change of coloration does not even have to be smooth, i.e. there can be breaking points. Different colormaps could be set up and immediately tested by using the Manhattan plot.

I tested a few other solutions for coloring using other colormaps. I have called two of the colormaps I tested: Manhattan night and Manhattan sunset. Both with a breaking point at 70%. In figures 7.1, 7.2 and 7.3 we can see the three plots tested on the same data.

The night and sunset colormaps further extent the suitability of the naming of the plot (though that is not really an important factor) and provides a guide-line of what high explained variance can be. However, because of the fact that few PCs will mostly be used and for its simplicity and cleanness, I would say, the standard gray-scale Manhattan plot is a better choice.

Issues

I chose to simply remove the attributes of X, for analysis, where there is STD = 0 for an attribute vector because of two reasons. Firstly, for normalization of the residual variable variances we end up with indefinite values (a result of $x/0$ in Python) and a choice would have to be made on how these data should be visualized in the plot. Secondly, when preprocessing

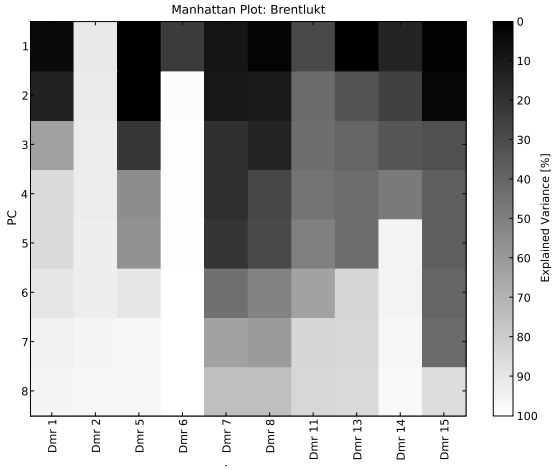

Figure 7.1: Using standard colormap in the plot, from black to white.

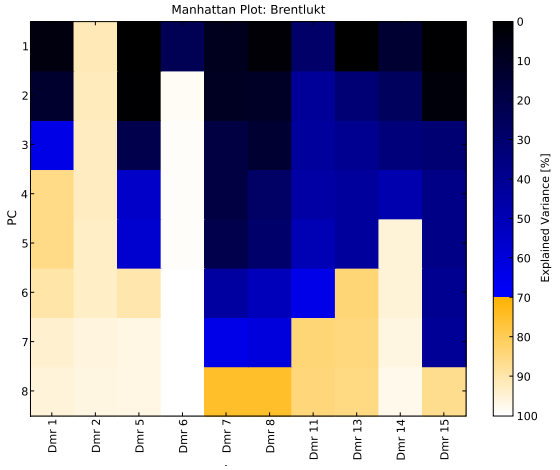

Figure 7.2: Using Manhattan night colormap in the plot, from black to white. Here bright colors still indicate high explained variance, but the curve is not smooth.

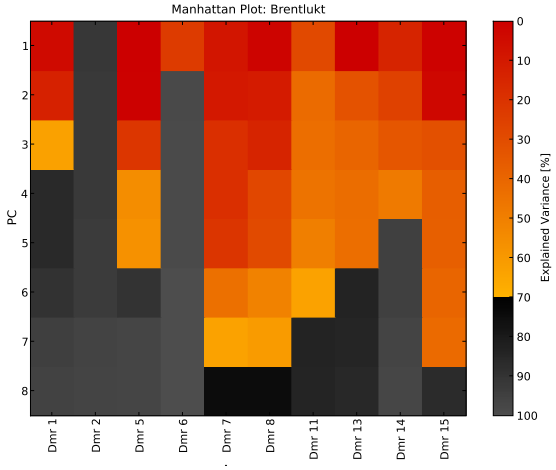

Figure 7.3: Using Manhattan sunset colormap in the plot, from a deep red to a dark gray. Here darkness indicate high explained variance.

include standardization of X it would not be possible to process the standardized data further, meaning something would have to be done with the standardized X. Both of these issues are solved by leaving these attribute vectors out of the analysis.

So in my opinion it was better just to remove the attributes from the analysis and give the user a message of what has been removed. This information is also available in the numerical data if any attributes are left out.

7.2 Performance

The Python SVD method of the NumPy package is interfaced to the LAPACK library [25]. A LAPACK lite version comes with NumPy as a C extension module (though LAPACK is built on Fortran).

I do not know the algorithm for computing eigenvectors used in NumPy's SVD method (possibly more than one depending on parameters given and different cases), but my assumption was that performance could be improved by implementing an iterative algorithm and enable the possibility of limiting PCs to compute. After all, it is not unusual that only the first few PCs of a PCA are of interest since those PCs explain most of the variance. Especially for the Manhattan plot it is suitable to limit the number of PCs to compute. I chose to implement NIPALS because it is an iterative algorithm. Additionally NIPALS is supposed to have high accuracy concerning results.

After implementing the NIPALS algorithm and testing that it worked it was interesting to see how it performed compared to NumPy's SVD method. The test-results is of a machine with Linux 2.6, an Intel Pentium IV 2.4GHz processor and 1GB RAM. Such hardware was new and powerful in 2003, but can now be considered weak. For newer hardware it is less important which PCA method we use.

The same is input is used for each method. The data is mean-centered, but not standardized. The scores, loadings and total explained variances (for each PC) are retrieved for each PCA. The number of PCs (lesser means fewer iterations using NIPALS) available after the PCA and the time used by the method is presented in these results:

```
PCA for each assessor of the cheese data set
All matrices have the shape (14 x 17):
=======================================
numpy.linalg.svd, 14 PCs, time:  20.0 ms
py-nipals matrix, 14 PCs, time: 930.0 ms
py-nipals array,  14 PCs, time: 950.0 ms
c-nipals,         14 PCs, time:  10.0 ms

py-nipals matrix,  4 PCs, time: 290.0 ms
py-nipals array,   4 PCs, time: 340.0 ms
c-nipals,          4 PCs, time:  10.0 ms
```

The first test results show PCA on the cheese data set, with averaged replicates. Manhattan data will usually be retrieved by a number of PCA on rather small matrices such as in the test-case above. In the following test-cases the data has been changed to larger matrices of random values.

```
1X PCA of matrix A (1000 x 200):
========================================
numpy.linalg.svd,  200 PCs, time:  2.420 s
py-nipals matrix,    4 PCs, time: 14.710 s
py-nipals array,     4 PCs, time:  6.470 s
c-nipals,          200 PCs, time:  8.110 s


1X PCA of matrix A (2000 x 100):
========================================
numpy.linalg.svd,  100 PCs, time:  8.770 s
py-nipals matrix,    4 PCs, time: 15.100 s
py-nipals array,     4 PCs, time: 12.880 s
c-nipals,          100 PCs, time:  4.700 s
```

```
10X PCA of matrix A (1000 x 1000):
==========================================
numpy.linalg.svd, 1000 PCs, time: 118.620 s
py-nipals matrix,    4 PCs, time: 765.490 s
py-nipals array,     4 PCs, time: 188.840 s
c-nipals,            4 PCs, time:  18.740 s
```

It is clear that compiled code, C-NIPALS or NumPy's SVD, should be used for PCA. The C-NIPALS PCA performs well in the cases of either very wide- or high-shaped matrices or generally where the number of PCs is to be limited to just a few (as e.g. with most cases of Manhattan plots). Other than that using NumPy's SVD for PCA is the best choice. However, there is one thing that is not covered in these results; all the E-matrices must be calculated after using NumPy's SVD, but as for NIPALS the E-matrices can be available immediately after the algorithm has been carried out. As mentioned earlier the E-matrices are necessary to retrieve the manhattan data.

Another point of interest of these results is the comparison of the NumPy matrix vs NumPy array in the Python implementations of NIPALS. I would have thought that in all cases the NumPy array method would outperform the NumPy matrix method, but it is not so for smaller matrices. This may be due to weaknesses of my NIPALS implementation using the NumPy array. However, for large matrices, there is no question on which of the two to use (see results using a 1000 x 1000 matrix).

7.3 Analysis

Multivariate analysis may be something difficult and complex, but exploration of the multivariate data becomes easier with such tools as PanelCheck.

Analyzing a plot of a PCA and its patterns is something that should be done by the human eye. Pattern recognition is something humans are very good at by nature, and it is very complicated to make "general" automated methods for evaluating such patterns. Simpler said, it is hard to train a computer to be skilled in pattern recognition.

7.3.1 PCA of The Cheese Data Set

As data matrix X for this PCA I have used the average matrix of all assessors of the cheese data set. Using PCA in this multivariate analysis may help in getting a conclusion on what sample (color of plastic on package) is best for storage of the cheese.

As mentioned in section 3.2.4 the first few PCs have the most explained variance. In fact, here PC1 has more than 90% explained variance. This

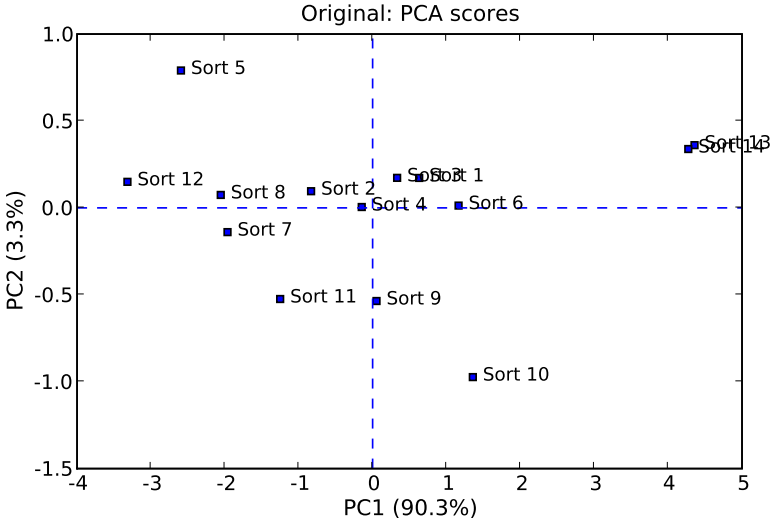

Figure 7.4: Score plot (PC1/PC2) for the consensus of the cheese data set.

indicates that almost all variance in the data is explained by PC1 alone. So when we are looking at these plots, we can mostly read of the horizontal PC axis (PC1). The variation explained by the remaining PCs is very low and might even represent only noise.

For creating these visualizations I have used PanelCheck and its Consensus plot. The Consensus plot uses averaged data, with average of each replicate for each sample, and carry out the PCA on all the assessors simultaneously (see figure 7.13).

An interesting thing we can see of these plots (see figure 7.4 and 7.5 on the next page), is that Syrllukt and Syrlsmak lies far to the left and Harsklukt and Harsksmak lies far out to the right (on the PC1 axis). We can see these attributes in the loading plot. Syrllukt and Syrlsmak is the acidic odor and flavor of a cheese (a sign of freshness), while Harsklukt and Harsksmak is the rancidity flavor and odor of a cheese (not good!). As we compare the two plots, this can give us the conclusion that there were something bad about the storage conditions in sample Sort 12, and something went right for Sort 13 and 14.

Here one can see by the plots that I have only chosen to do the analysis with only PC1 and PC2. To answer why I will again refer to section 3.2.4 about residuals (E). There is often a breaking point such as seen in figure 7.6 on page 73. This will give you an indication on how many PCs to use. The simple solution is to look for a breaking point and include as many PCs there are up until that breaking point. Such a breaking point often occurs for that plot, but it could have been a smooth curve also. A

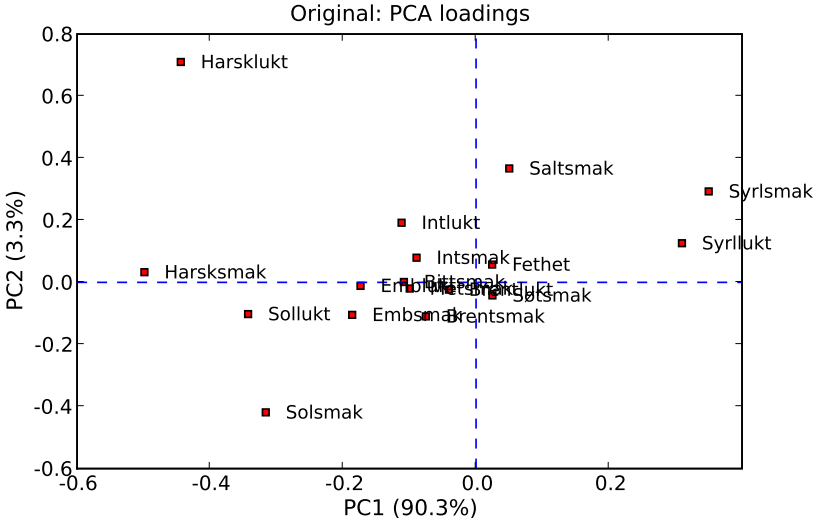

Figure 7.5: Loading plot (PC1/PC2) for the consensus of the cheese data set.

smooth curve can be the result of an analysis without structure (very little correlation between any of the attributes). By looking at figure 7.6 on the next page it suggests that one should use only PC1, but to be able create a 2-dimensional plot and for possibly finding further grouping indications, I chose to include PC2 also.

Already this has been a short example of exploratory data analysis where our conclusions are that we should choose sample 13 or 14 for better storage and longer lasting freshness of the cheese. See table 2.1 on page 12 for information on the samples.

PCA of Individual Assessors

We may do PCA on individual assessors. This gives us more details for those assessors, their individual performance and we can compare them to the consensus. For comparison, in this example, assessor 1 and 12 have been chosen, to compare one assessor with much experience and one without.

Firstly, we can see that assessor 1 has high explained variance for PC1 (see figure 7.7), similar to that of the consensus. We can again see, by comparing the score- and loading-plot (see figures 7.7 and 7.8), that sample 13 and 14 are mapped by attributes Syrsmak and Syrlukt, which indicate freshness. Harsksmak, Harsklukt, Sollukt and Solsmak are negative attributes and lie far to the left, the opposite of Syrsmak and Syrlukt which are positive attributes and lie far to the right, on PC1 axis. PC1 has most

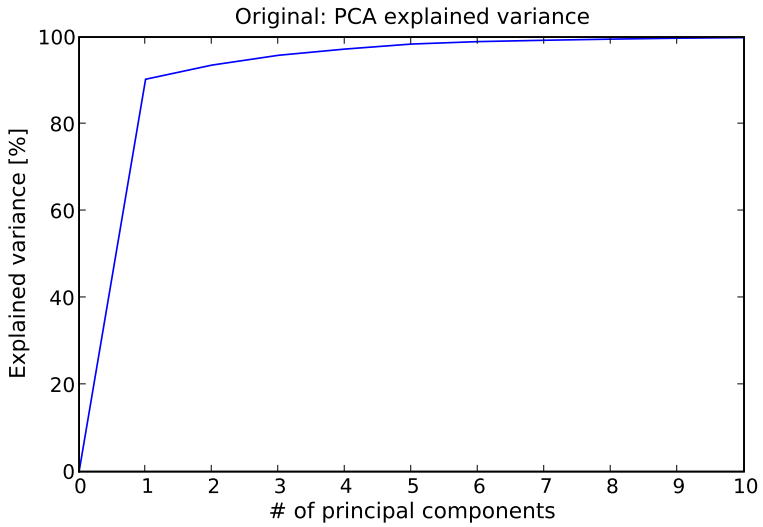

Figure 7.6: Explained variances for the consensus of the cheese data set.

explained variance (92.2%), which make the attributes mentioned the most important in this case. Actually we would come out with similar conclusions of the results for assessor 1 alone as with the consensus.

Assessor 12 does not have high explained variance for PC1 (36.5%) and the patterns seen of score- and loading-plots (see figures 7.9 and 7.10) are quite different from that of the consensus and assessor 1. For PC4 (11.6%) we can start to see some of the same indications concerning attributes Syrsmak vs Harsksmak and samples 12, 13 and 14. But the explained variance is very low for these attributes and we may conclude that the data is too noisy and that the assessor needs more training.

Finally, in figures 7.11 and 7.12, we can see the explained variances for each PC presented for assessor 1 and 12. Again we can see that assessor 1 agrees more with the consensus and that assessor 12 has very low explained variance, in fact we do not pass the 90% barrier until PC9.

7.3.2 Manhattan Analysis of The Cheese Data Set

In the preceding section we were looking at the consensus of the assessors and afterwards we compared assessor 1 and 12. But instead of comparing assessors one by one, of various score- and loading-plots and using different PCs, we may get some of the same conclusions much faster by using just a few Manhattan plots (or possibly only by one overview plot). Manhattan plot is made for investigating results of individual assessors. Figure 7.13 illustrate the setup of the data for PCA.

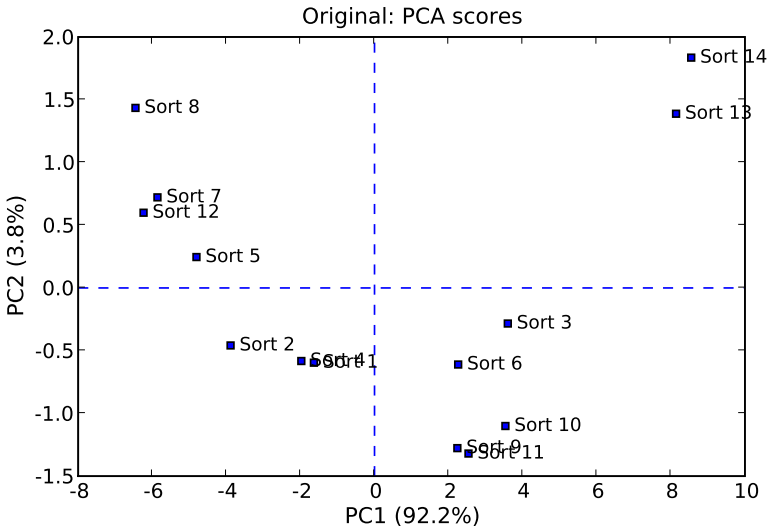

Figure 7.7: Score plot (PC1/PC2) for assessor 1 of the cheese data set.

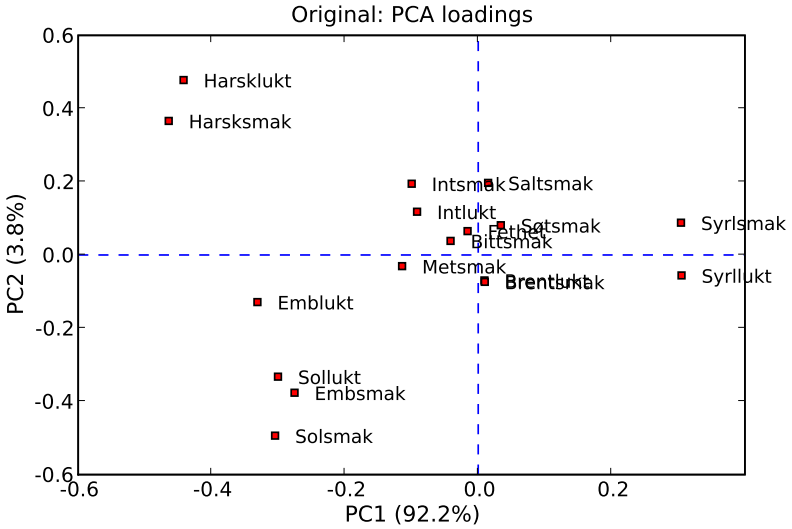

Figure 7.8: Loading plot (PC1/PC2) for assessor 1 of the cheese data set.

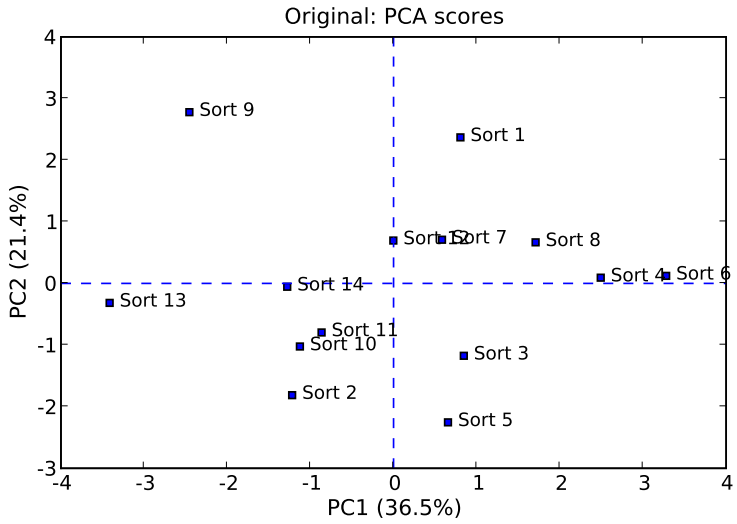

Figure 7.9: Score plot (PC1/PC2) for assessor 12 of the cheese data set.

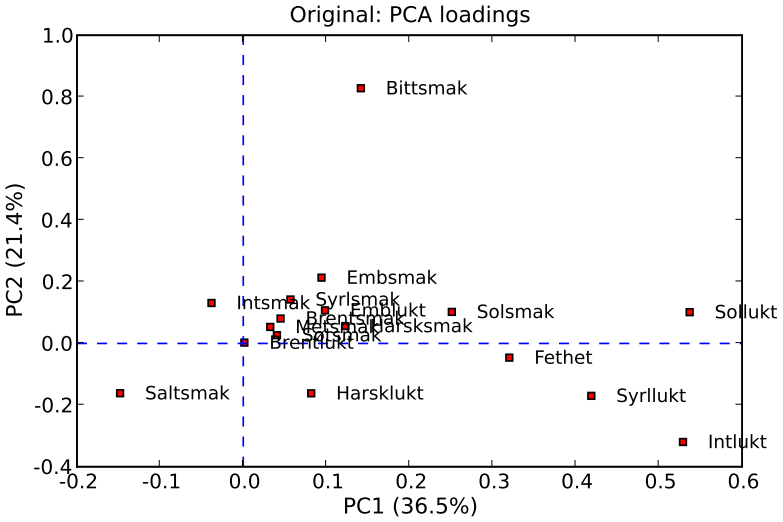

Figure 7.10: Loading plot (PC1/PC2) for assessor 12 of the cheese data set.

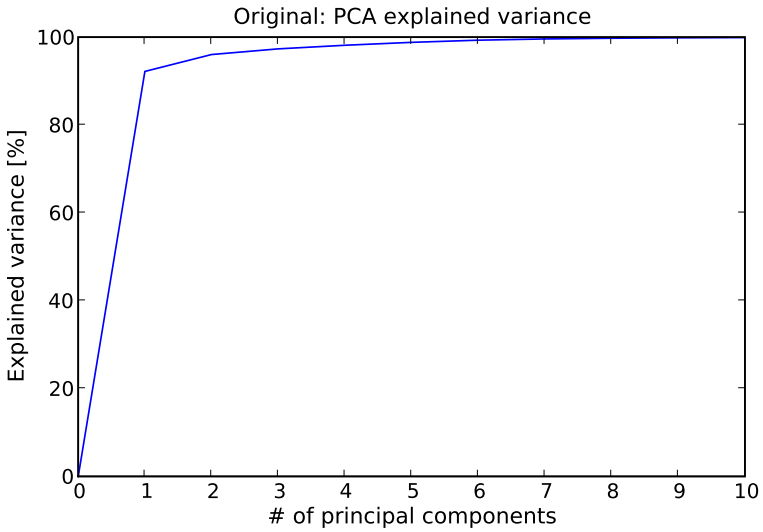

Figure 7.11: Explained variances for assessor 1 of the cheese data set.

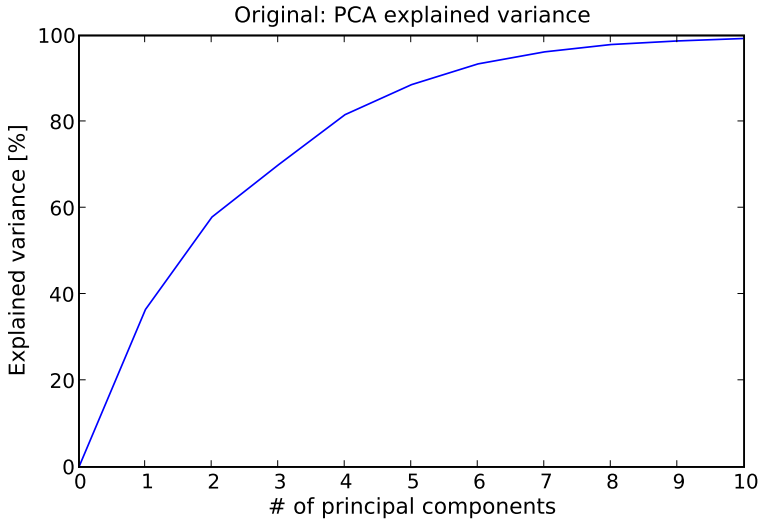

Figure 7.12: Explained variances for assessor 12 of the cheese data set.

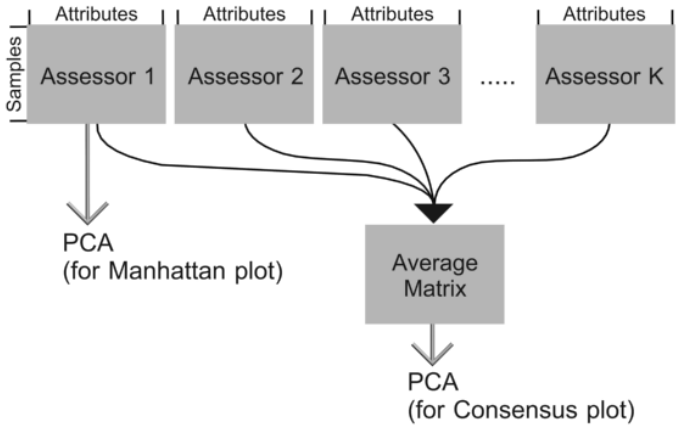

Figure 7.13: Illustration of setup of data matrix for PCA of Manhattan plot and Consensus plot.

With the Manhattan plot we can see deeper into to the data set and not only compare two or possibly three PCs at a time. Score- and loading-plots in PanelCheck can only be visualized by two dimensions (PCs) at a time.

Two things we cannot see by using Manhattan plots are distribution of samples and information on how attributes contribute to distribution of variance, even though this information is something that can be retrieved of a PCA.

When using Manhattan plot we will be looking for darkness, actually, the low explained variable variances and change of explained variance between PCs. Darkness in the Manhattan plot means low explained variance (noisy data) for that particular attribute. If two assessors have given exactly the same score values, on one attribute for all samples, their levels of explained variance, and thus their shades of gray in the Manhattan plot, would be equal for that particular attribute.

For this Manhattan analysis of the cheese data set we will, again, be comparing assessors 1 and 12. First, let us take a look at the Manhattan plot for assessor 1 (see figure 7.14). Assessor 1 shows a systematic variance for the attributes, if we compare with other assessors, and with around 90% explained variance already for PC1 on many of the attributes (Syrlukt, Sollukt, Emblukt, Harsklukt, Syrlsmak, Solsmak, Metsmak, Embsmak and Harsksmak) again in agreement with other assessors. Where there is unexplained variance for assessor 1, most of the other assessors also have unexplained variance, this can be seen for attributes Søtsmak, Saltsmak and Fethet (sweet flavor, salt flavor and fattiness) of figure 7.16 (plot numbers 7, 8, and 12). We can see which attributes that are important and at what PC

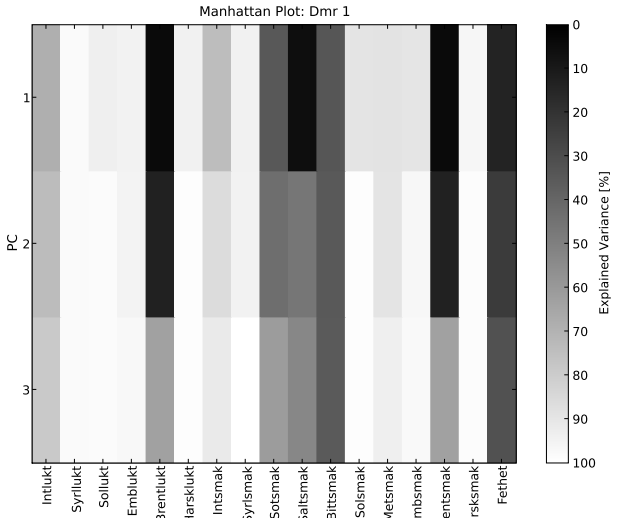

Figure 7.14: Manhattan plot for assessor 1 of the cheese data set.

they are explained well, and investigate further with score- and loading-plots.

For assessor 12 the Manhattan plot looks darker, literally (see figure 7.15). Darkness does not necessarily have to be something negative, but when comparing with other assessors and considering the actual attributes of the test we can see less systematic variance for assessor 12. This indicates noisy data. Randomized score values on attributes would produce very noisy data, resulting in dark Manhattan plots.

The highly likely reason for the differences between assessor 1 and assessor 12 is that assessor 1 has worked in sensory analysis for a long time while assessor 12 had just started as assessor and had little experience.

The assessors show agreement for some attributes with low explained variance, as mentioned. In figure 7.16 we can see a clear difference of which attributes that are hard to explain and others that are explained well. Additionally we see differences between the trained and the untrained group. The three last assessors are untrained.

One example of an attribute that is hard to explain is "fethet" (see the last Manhattan plot of figure 7.16), which means fattiness. This is reasonable because all the samples are of one type of cheese (produced in the same batch). If there had been many types of cheeses involved we could have expected more variance between the different samples and possibly higher explained variance for that attribute in the Manhattan plot. This can also be the general conclusion, where there is low explained variance for attributes and at the same time agreement among the assessors, that they simply can-

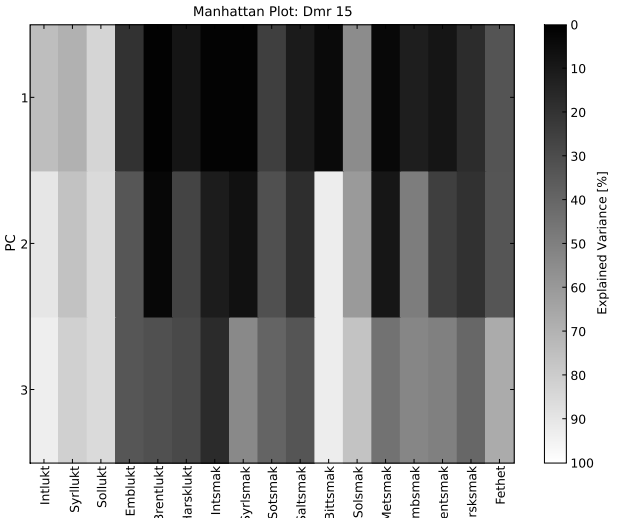

Figure 7.15: Manhattan plot for assessor 12 of the cheese data set.

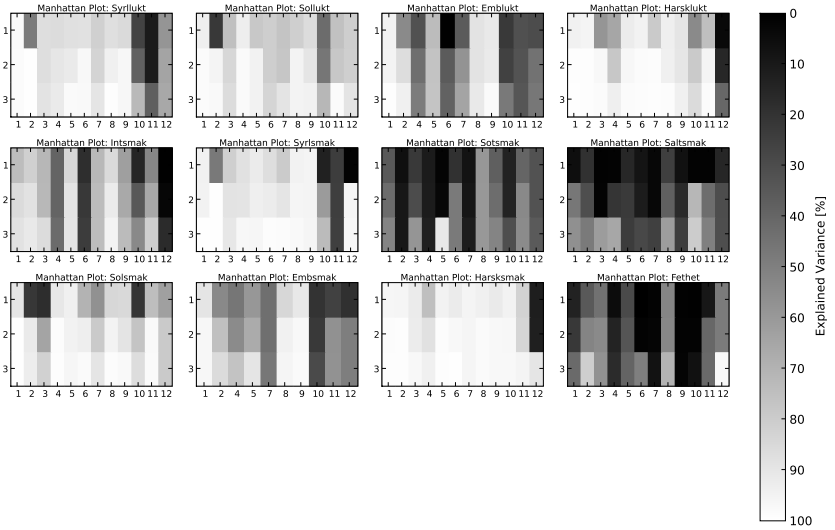

Figure 7.16: Manhattan overview plot for selected attributes of the cheese data set.

not sense differences between the samples (which can mean there are no differences).

Conclusions of the results seen here with the Manhattan plot is that assessor 12 needs more training. By leaving the assessors that needs more training out of the consensus PCA may lead to more accurate results. In this case leaving the three untrained assessors out of the PCA increases the explained variance to 91.7% from 90.3% for PC1.

Improving the consensus is useful, but generally the main focus of the Manhattan plot is the ability to investigate structure of variance for individual assessors for training purposes.

Chapter 8

Conclusion

In this thesis PanelCheck has been introduced, a software tool to carry out analysis with statistical methods on sensory data. PanelCheck creates 2-dimensional visualizations for analysis and makes raw data and numerical data of calculations available for each statistical method. The main assignment of this thesis has been to implement the Manhattan plot in PanelCheck as a standard plot module. I have gone through software tools used in PanelCheck for the purpose of getting started on usage. Then described the design of PanelCheck to introduce developing for PanelCheck, specifically on how to create a new plot module.

The Manhattan plot may be used as an alternative way to view explained variance of a PCA. It provides the user with the ability to investigate explained variable variances for individual assessors on more than two PCs of the PCA simultaneously. PCA is carried out for each assessor and a plot of the explained variance for each attribute can be created. Another way of projecting the same data has been implemented such that the cumulative explained variances are sorted by attribute and not by assessor as mentioned above. In this way the user can easily identify attributes where the performance of the assessors varies much within the panel. Manhattan plot can be used as a rapid screening tool for detection of problems in performance. A module for Python has been created to carry out PCA.

In this implementation of the plot a rectangular net of polygons (quads) is created and each polygon is colored by the level of cumulative explained variance accordingly. This is done by using matplotlib with wx as backend.

Already it is proving to be a valuable addition to PanelCheck based on feedback from testers. A test-version of PanelCheck 1.3.0 has been under testing at Matforsk and at some of the partners of the industry since earlier in 2008. One minor improvement that has been implemented, by suggestions of the testers, is that the colorbar of the plot has been flipped vertically

to match the cumulativeness of the explained variance in the plot.

Research Version

Concerning software development I believe many of the concepts in PanelCheck work well, often simple and straight-forward with usage of nice features of Python. But one area that should be improved further is the implementation of a new plot module. By improving usage of OOP, of course with reusability in mind, sections such as Exporting and PlotView could automatically be completely functional much easier. After creation of a new plot module it should be possible to create its tab, in the GUI, and the tab-tree without having to deal with wxPython (or the current graphics-toolkit) directly. Additionally code should be cleaned up. All this is important, especially, since there is interest in work on a research version of PanelCheck. Furthermore, the PanelCheck code needs more and throughout consistent documentation.

Appendix A

Word List

ANOVA
ANalysis Of VAriance between groups.

Backend
Computer software related a backend is a system that drives a frontend. An example is: the kernel of an operating system. An example of a frontend is: the GUI in an operating system.

Consensus
General agreement among a group.

Correlation
In statistics, correlation (often measured as a correlation coefficient), indicates the strength and direction of a linear relationship between two random variables.

Covariance
The measure of how much two random variables vary together (as distinct from variance, which measures how much a single variable varies).

Multivariate
Multidimensional, there exists a number of independent variables.

Multivariate analysis
In statistics it describes a collection of procedures which involve observation and analysis of more than one statistical variable at a time.

NIPALS
The "Nonlinear Iterative Partial Least Squares" algorithm, for finding eigenvectors.

OOP
Object Oriented Programming, a fundamental programming style that

uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as, encapsulation, modularity, polymorphism, and inheritance.

STD
Standard Deviation, in statistics, of a probability distribution, random variable, or population or multiset of values is a measure of the spread of its values.

Univariate
When a statistical technique is to be used, it contains only one dependent variable.

Variance
In statistics, the variance of a random variable, probability distribution, or sample is one measure of statistical dispersion, averaging the squared distance of its possible values from the expected value.

Appendix B

Source Code

B.1 SensoryData Class

```
#!/usr/bin/env python

from numpy import average, array, ndarray, vstack, zeros, transpose

class SensoryData:
    def __init__(self, abspath):
        """
        The SensoryData class for PanelCheck

        @type abspath:  string
        @param abspath: Absolute file-path for the data file to be read.
        """
        # absolute file path
        self.abspath = abspath
        # file name
        self.filename = ""
        # character codec set, extended latin-1  (might be changed during data load)
        self.codec = "mbcs"

        self.scale_limits = () # (x_min, x_max, y_min, y_max)
        self.real_limits = () # (x_min, x_max, y_min, y_max)

        # nametag lists
        self.AssessorList = []
        self.SampleList = []
        self.ReplicateList = []
        self.AttributeList = []

        # column indices:
        self.ass_index = 0
        self.samp_index = 1
        self.rep_index = 2
        self.value_index = 3 # where score-values start

```

```
35          self.LabelList = []
36          self.ListCollection = []
37          self.Matrix = None
38
39          # dictionary, self.SparseMatrix[(assessor, sample, replicate)]
40          # will point to ndarray (numpy array)
41          self.SparseMatrix = {}
42
43      def GetAssessorAveragedDataAs2DARRAY(self, active_samples=None, \
44                                                 active_attributes=None, \
45                                                 active_replicates=None, \
46                                                 active_assessor=None):
47          """
48          Returns data matrix of active data for given assessor.
49
50          @type active_samples: list
51          @param active_samples: list of strings (active sample names)
52
53          @type active_attributes: list
54          @param active_attributes: list of strings (active attribute names)
55
56          @type active_replicates: list
57          @param active_replicates: list of strings (active replicate names)
58
59          @type active_assessor: string
60          @param active_assessor: the current assessor (name)
61
62          @return: numpy array (2-dimensional active data for current assessor)
63          """
64
65          if active_samples == None: active_samples = self.SampleList
66          if active_attributes == None: active_attributes = self.AttributeList
67          if active_replicates == None: active_replicates = self.ReplicateList
68
69          if active_assessor == None:
70              return None # no active assessor
71
72          att_indices = []
73          for att in active_attributes:
74              # the same order as in self.SparseMatrix:
75              att_indices.append(self.AttributeList.index(att))
76
77          m_data = zeros((len(active_samples), len(active_attributes)), float)
78          m_data_ind = 0
79          reps_data = zeros((len(active_replicates), len(active_attributes)), float)
80
81
82          for samp in active_samples:
83              rep_ind = 0
84              for rep in active_replicates:
85                  row_data = self.SparseMatrix[(active_assessor, samp, rep)]
86
87                  # fill one row in m_data:
88                  for ind in range(len(active_attributes)):
```

```
                    reps_data[rep_ind, ind] = row_data[att_indices[ind]]
                rep_ind += 1

            m_data[m_data_ind] = average(reps_data, 0)
            m_data_ind += 1

        return m_data
```

B.2 PlotData Class

```
#!/usr/bin/env python

class PlotData:
    """
    Class containing settings for a general plot.
    """
    def __init__(self, active_ass=[], active_att=[], active_samp=[], \
                       tree_path=[], view_grid=False, view_legend=False):
        # gui selections:
        self.activeAssessorsList = active_ass
        self.activeAttributesList = active_att
        self.activeSamplesList = active_samp
        self.tree_path = tree_path # path selection in tree (old itemID)
        self.view_grid = view_grid
        self.view_legend = view_legend
        self.overview_plot = False
        self.aspect = 'auto'

        # standard plot data:
        self.error = -1 # -1 (not checked), 0 (no errors), 1 (errors)
        self.fig = None # matplotlib Figure
        self.ax = None # matplotlib Axes
        self.limits = [0.0, 1.0, 0.0, 1.0] # axes limits [xmin, xmax, ymin, ymax]

        self.raw_data = [] # raw data scores
        self.numeric_data = [] # numerical data of the calculations

        self.point_lables = []
        self.point_lables_type = 0 # 0: points, 1: lines
        self.plot_type = "general plot" # plot identification (name)

    def actives_changed(self, active_ass=[], active_att=[], active_samp=[]):
        """
        Returns True if active-lists of assessors, attributes
        or samples have been changed. If there is no change it returns False.
        """
        if not self.activeAssessorsList == active_ass: return True # actives changed
        if not self.activeAttributesList == active_att: return True # actives changed
        if not self.activeSamplesList == active_samp: return True # actives changed
        return False # no change
```

Bibliography

[1] K. Esbensen: *Multivariate Analysis - in practice*, CAMO, 3rd edt. (1998)

[2] K. V. Mardia, J. T. Kent and J. M. Bibby: *Multivariate Analysis*, Academic Press, (1979)

[3] T. Dahl, O. Tomic, J.P. Wold and T. Næs: *Some new tools for visualising multi-way sensory data*, Research article, (2007)
`http://www.sciencedirect.com/`

[4] O. Tomic, A. Nilsen, M. Martens and T. Næs: *Visualization of sensory profiling data for performance monitoring*, LWT 40 (2007), p262-269

[5] G. Strang: *Introduction to Linear Algebra*, Wellesley-Cambridge, 2nd edt. (1998), p245-271, p313-321

[6] E. Risvik and T. Næs: *Multivariate Analysis of Data in Sensory Science*, Elsevier Science, (1996)

[7] H. Martens and T. Næs: *Multivariate Calibration*, John Wiley & Sons, (1991)

[8] H. Lohninger: *Teach/Me Data Analysis*, Springer-Verlag, (1999)
`http://www.vias.org/tmdatanaleng/dd_nipals_algo.html`

[9] L. R. Tucker: *The extension of factor analysis to three-dimensional matrices*
In N. Frederiksen & H. Gulliksen: *Contributions to Mathematical Psychology*, New York: Holt, Rinehart & Winston, (1964), p110-182

[10] H. Abdi and D. Valentin: *The STATIS Method*
In N. J Salkind: *Encyclopedia of Measurement and Statistics*, Thousand Oaks (CA): Sage, (2007), p955-962
`http://www.utdallas.edu/~herve/Abdi-Statis2007-pretty.pdf`

[11] P. B. Brockhoff: *Statistical testing of individual differences in sensory profiling, Food Quality and Preference*, (2003), p425-443

[12] The PanelCheck Project
`http://www.matforsk.no/panelcheck/`

[13] PanelCheck Source Code
`https://sourceforge.net/projects/sensorytool/`

[14] PCA Module for Python
`http://folk.uio.no/henninri/pca_module/`

[15] W. Savitch: *Absolute C++*, Addison-Wesley, 1st edt. (2002)

[16] C Language Library
`http://www.cplusplus.com/reference/clibrary/`

[17] H. P. Langtangen: *Python Scripting for Computational Science*, Springer-Verlag, 2nd edt. (2006)

[18] Python Programming Language
`http://www.python.org/`

[19] T. Peters: *The Zen of Python*, Poetry, (2004)

```
>>> import this
```

[20] Python Books
`http://wiki.python.org/moin/PythonBooks/`

[21] G. van Rossum: *Python/C API Reference Manual*, Python Software Foundation, Release 2.5.2, (2008)
`http://docs.python.org/api/`

[22] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin and T. E. Oliphant: *Numerical Python*, (2001)
`http://numpy.scipy.org/numpydoc/numpy.html`

[23] T. E. Oliphant: *Guide to NumPy*, Trelgol Publishing, (2006)
`http://numpy.scipy.org/`

[24] SciPy Package Documentation
`http://www.scipy.org/Documentation/`

[25] LAPACK – Linear Algebra PACKage
`http://www.netlib.org/lapack/`

[26] MATLAB® - The Language of Technical Computing
`http://www.mathworks.com/products/matlab/`

[27] J. Hunter and D. Dale: *The Matplotlib Userś Guide*, Matplotlib 0.90.0 user's guide, (2007)
`http://matplotlib.sourceforge.net`

[28] The AGG Project
http://antigrain.com/

[29] wxPython GUI toolkit
http://www.wxpython.org/

[30] wxWidgets Project
http://www.wxwidgets.org/

[31] Simplified Wrapper and Interface Generator (SWIG)
http://www.swig.org/

[32] Python Win32 Extensions
http://sourceforge.net/projects/pywin32/

[33] RPy (R from Python)
http://rpy.sourceforge.net/

[34] The R Project for Statistical Computing
http://www.r-project.org/

[35] GNU General Public License, v. 2, (1991)
http://www.gnu.org/copyleft/gpl.html

[36] Wikipedia: Unit Testing
http://en.wikipedia.org/wiki/Unit_test/

[37] *ISO (1983) Sensory analysis - Methodology - Flavour profile methods*
ISO 6564:1985